Table of Contents

Introduction 1.1
Licence 1.2
Foreword 1.3
Credits 1.4
Notation used through this book 1.5
Setting Up Rust 1.6
C and C++ Background 1.7
Rust Background 1.8
Let's Start Simple 1.9
Compiling and Linking in More Detail 1.10
Source Layout and Other General Points 1.11
Namespacing With Modules 1.12
Porting Code 1.13
Features of Rust compared with C++ 1.14
- Types 1.14.1
- Strings 1.14.2
- Variables 1.14.3
- Literals 1.14.4
- Collections 1.14.5
- Structs 1.14.6
- Comments 1.14.7
- Lifetimes, References and Borrowing 1.14.8
- Expressions 1.14.9
- Conditions 1.14.10
- Switch / Match 1.14.11
- Casting 1.14.12
- Enumerations 1.14.13
- Loops 1.14.14
- Functions 1.14.15
- Polymorphism 1.14.16

Error Handling 1.14.17
Lambda Expressions / Closures 1.14.18
Templates / Generics 1.14.19
Attributes 1.14.20
Multi-threading 1.14.21
Lint 1.14.22
Macros 1.14.23
Memory Allocation 1.14.24
Foreign Function Interface 1.14.25
Porting from C/C++ to Rust 1.15
Copy Constructor / Assignment Operators 1.15.1
Missing Braces in Conditionals 1.15.2
Assignment in Conditionals 1.15.3
Class Member Initialisation 1.15.4
Headers and Sources 1.15.5
Forward Declarations 1.15.6
Namespace Collisions 1.15.7
Macros 1.15.8
Type Mismatching 1.15.9
Explicit / Implicit Class Constructors 1.15.10
Poor Lifetime Enforcement 1.15.11
Memory Allocation 1.15.12
Null Pointers 1.15.13
Virtual Destructors 1.15.14
Exception Handling / Safety 1.15.15
Templates vs Generics 1.15.16
Multiple Inheritance 1.15.17
Linker Errors 1.15.18
Debugging Rust 1.16
Memory Management 1.17
Rust's std:: library 1.18
Rust Cookbook 1.19

A Guide to Porting C/C++ to Rust

This book is for people familiar with C or C++ who are thinking of using Rust.

Before we go into what Rust is or why it might be preferable to C/C++ *in some cases*, let's think of software that is mission critical and must not or should not fail.

- Operating system services and daemons
- Internet of things devices
- Industrial control software
- Medical devices - MRI, ultrasound, X-ray, ventilators etc.
- High availability servers / databases / cloud storage etc.
- Avionics, telemetry, rocketry, drones etc.

All this code must run as efficiently and reliably as possible. It must run on devices for days, weeks, months or preferably years without failure. It cannot suffer intermittent freezes, erratic performance, memory leaks, crashes or other issues without impacting on its purpose.

Normally such software would be written in C or C++, but consider these *every day* programming issues that can afflict these languages:

- Dangling pointers. A program calls an invalid pointer causing a crash.
- Buffer overruns / underruns. Code writes beyond an allocated buffer causing memory corruption or a page exception.
- Memory leaks. Code that allocates memory *or resources* without calling the corresponding free action. C++ provides classes such as smart pointers and techniques like RAII to mitigate these issues but still occur.
- Data races. Multiple threads write to data at the same time causing corruption or other destabilizing behavior.

Rust stops these bad things happening **by design**. And it does so without impacting on runtime performance because all of these things are checked at compile time:

- Object lifetimes are tracked automatically to prevent memory leaks and dangling pointers.
- The length of arrays and collections is enforced.
- Data race conditions are prevented by strict enforcement of mutex / guards and object ownership.

Code that passes the compiler's checks is transformed into machine code with similar performance and speed as the equivalent C or C++.

This is a "zero-cost" approach. The compiler enforces the rules so that there is zero runtime cost over the equivalent and correctly written program in C or C++. Safety does not compromise performance.

In addition Rust plays well C. You may invoke C from Rust or invoke Rust from C using foreign function interfaces. You can choose to rewrite a critical section of your codebase leave the remainder alone.

For example, the Firefox browser uses Rust to analyse video stream data - headers and such like where corrupt or malicious code could destabilize the browser or even be exploitable.

Why Rust?

Let's start by saying if what you have works and is reliable, then the answer to the question is "there's no reason" you should consider porting.

However if you have code that *doesn't* work or *isn't* reliable, or *hasn't* been written yet or is due a major rewrite then perhaps you have answered your own question.

You could write the code or fixes in C/C++ in which case you have to deal with all the unsafe issues that the language does not protect you from. Or you might consider that choosing a safe-by-design language is a good way to protect you from suffering bugs in the field when the code is supposed to be ready for production.

Rust is not a magic wand

Despite the things the language can protect you against, it cannot protect you against the following:

- General race conditions such as deadlocks between threads
- Unbounded growth, e.g. a loop that pushes values onto a vector until memory is exhausted.
- Application logic errors, i.e. errors that have nothing to do with the underlying language, e.g. missing out the line that should say "if door_open { sound_alarm(); }"
- Explicit unsafe sections doing unsafe and erroneous things
- Errors in LLVM or something outside of Rust's control.

Licence

The book is written under these terms:



Refer to the link for the exact legal terms. But in essence you may share and modify this book providing you do not sell or derive profit from doing so.

Foreword

BEGIN DRAFT BOOK DISCLAIMER

- Some of the samples will not compile or may not have been syntax checked
- C and Rust code snippets are not distinguished very well yet (styling)
- Some of the text makes uncited assertions of fact
- Some of the text is marked TODO
- Some of the topics that should be covered are brushed over, given undue weight or omitted entirely
- Some of the text probably makes no sense or repeats itself

WITH ALL THAT IN MIND, read on!

END DRAFT BOOK DISCLAIMER

Think of all the software that needs to be reliable in this world. Software that can ill afford downtime or crashes. Software that is mission critical and must not or should not fail.

- Operating system services and daemons
- Internet of things devices
- Industrial control software
- Medical devices, imagery etc.
- High availability servers / databases / cloud storage etc.
- Avionics, telemetry, rocketry, drones etc.

All this code that has to run as efficiently and reliably as possible with the minimal of errors. It also has to be predictable without sudden freezes or mystery-memory behavior due to garbage collection.

C and C++ has the speed angle covered but is hard to make reliable. A language like Java would have the reliability angle covered but is hard to make performant.

What we want is something which runs as fast as C or C++ but has the reliability that goes with it. And that is what Rust is about. It compiles into binary executables or libraries just like C or C++ and can even be used to produce dynamic libraries that can be consumed by other code bodies.

All of the information found in this document can be gleaned from elsewhere but it tends to be scattered across documents and sites that are focused on topics not and in many cases I've sought inspiration and knowledge from TODO

1. Online documentation for Rust
2. The Rustonomicon - describes some of the esoteric reasoning and internals behind the language and how to perform unsafe programming. Unsafe programming is not the default but is necessary for interacting with external code.
3. TODO - Design patterns repo
4. TODO - Effective C++, Meyers
5. TODO – various stackoverflow questions and answers

Notation used through this book

Code samples are given throughout this book are for C, C++, Rust and general configuration / console output.
In order to distinguish each kind they are styled as follows:

C / C++ samples are given in this style:

```
// C/C++
while (x < y) {
  cout << "x is less than y" << endl;
  ++x;
}
```

Rust samples are given in this style:

```
// Rust
if x == 20 {
  println!("Warning!");
}
```

Standard console output or script is given this style:

```
cd myproject/
cargo build
```

Most of the code samples are abbreviated in some fashion. e.g. they assume the code is running from within a main() function or they omit noise such as #includes, namespace definitions and so forth.

Setting up Rust

This section will talk you through setting up Rust for the first time and also how to keep it up to date.

Getting started is incredibly easy but some details vary upon your target platform. Rust runs on Windows, Linux and MacOS. In addition you might wish to cross-compile code for the consumption of another platform.

Use Rustup

The easiest way to get started is to download and run `rustup-init` which you can do by visiting the Rustup site.

The instructions differ for Windows and Unix-like systems:

- On Windows, rustup-init is an exe installer.
- On Unix / OS X / Linux rust-init is a shell script.

Either way, when you follow the instructions the installer will download and install put rustc, cargo, and rustup in your bin directory which is `~/.cargo/bin` on Unix and `%USERPROFILE%.cargo.\bin` on Windows.

It will also set your `PATH` environment variable so that you can open a terminal and type rustc, cargo, rustup etc.

Once `rustup` is installed you can also use the tool for maintenance:

- Install additional Rust toolchains (e.g. if you are cross-compiling or supporting multiple targets you may have more than one toolchain)
- Change the default toolchain that is invoked when you type `rustc` or `cargo`. Rustup will create symbolic links / scripts that invoke the appropriate toolchain
- Update the toolchain when a new version of Rust is released
- Fetch source and documentation

Unix / Linux

The process for running `rustup-init.sh` is as follows:

1. Open a terminal / console
2. Type "curl https://sh.rustup.rs -sSf | sh"

3. This will download and execute a script which will examine your environment, recommend the toolchain to download, and offer to modify your `PATH` environment variable.
4. Choose the option 1 to proceed. Or customize if you want to modify something
5. Wait for download to complete
6. You're done.

If you don't have curl, then you must install it first to proceed, or save the shell script from a browser to disk and execute that.

To install `curl` in Linux you would invoke a command like this to install it.

- Debian / Ubuntu - `sudo apt get curl`
- Fedora / Redhat - `sudo dnf install curl`

Windows

1. Download rustup-init.exe from rustup.rs.
2. Double click on the rust-init.exe and a console will open
3. Choose the option 1 to proceed. Or customize if you want to modify something
4. Wait for download to complete
5. You're done.

If you prefer not to go with the defaults, here are some choices you should decide upon:

1. 32/64 bit version. Most Windows installations are going to be 64-bits these days but you may have a reason to choose 32-bit.
2. GNU or MSVC ABI. This depends on what toolchain and runtimes you wish to be compatible with.

The second choice concerns the application binary interface (ABI) you want Rust to be compatible with.

- If you don't care about linking to anything then choose the GNU ABI. Also choose it if you have DLLs produced by MingW / MSYS. The advantage of this ABI is that it is more mature.
- If you have Visual Studio installed or intend to use Rust against DLLs created with Visual Studio, that's the ABI you need. One advantage of this option is that you can debug Rust inside of Visual Studio- the compiler will produce .pdb files that allow you to step debug Rust.

Keeping Rust up to date

New versions of Rust appear in a semi-frequent basis. If you want to update your environment to the latest version, it is as simple as this:

```
rustup update
```

Sometimes rustup will get an update of its own in which case you type:

```
rustup self update
```

Adding Rust source

Rustup installs a rust toolchain but if you're writing code or debugging you probably should also get the Rust source code so you can step into it or look at the implementation:

```
rustup component add rust-src
```

Manual installation

If you prefer manual installation of Rust then there are packages and instructions on the Rust site.

Just be aware that Rust has a fairly rapid release cycle so you probably only want to do this if you have a reason to choose a specific version of Rust and stick with it.

Otherwise you may find yourself uninstalling and reinstalling a new version 6 weeks later all over again.

Setting up a debugger

Unix / Linux

Debugging Rust is little different from debugging C or C++.

You must install gdb for your platform and then you may invoke it from a console or your favourite front-end to debug Rust code.

On Linux systems you would normally install gdb from a package with one of these commands:

```
sudo apt-get install gdb
# or
sudo dnf install gdb
```

You may also prefer to use lldb which is a companion project to LLVM (the backend compiler used by Rust). Refer to the lldb website for information on using it.

Rust comes with a few scripts that wrap gdb and lldb to provide pretty-printing to assist with debugging. When debugging, you can invoke `rust-gdb` or `rust-lldb` to use them.

Windows

If you have chosen Rust with the MSVC ABI then you can debug through Visual Studio with some limitations. When you create a debug build of your code, the compile will also create a .pdb file to go with it. You may open your executable in Visual Studio and step debug it, inspect variables and so on.

GDB

GDB on Windows is available through MSYS / MingW distributions.

For example downloads of the TDM-GCC distribution of MSYS can be found here. At the time of writing this, there is a standalone gdb-7.9.1-tdm64-2.zip containing the Choose the 32 or 64-bit version according to your Rust environment.

Extract the zip file to a directory, e.g. `C:\tools\gdb-7.9.1-tdm64-2` and add a value to your `PATH` environment variable:

```
set PATH=%PATH%;C:\tools\gdb-7.9.1-tdm64-2\bin\
```

You can invoke `gdb` from the command line but more normally you'd prefer a front end.

At the time of writing, perhaps the best option is Visual Studio Code which has plugins for debugging with GDB and for Rust development. So you can edit and debug from the same IDE.

Pretty printer

Rust supplies a pretty printer for variable inspection that you can add to the GDB. The pretty printer is a script written in Python that GDB will invoke to display variables.

First ensure you have Python 2.7 installed in your path.

The script is bundled with the Rust source code so you need to have installed that first.

If you installed it with `rustup` then it can be found in your `%USERPROFILE%\.rustup` directory:

e.g.

```
c:\users\MyName\.rustup\toolchains\stable-x86_64-pc-windows-gnu\lib\rustlib\src\rust\src\etc
```

Otherwise it can be found wherever you unzipped your Rust source code under `src\rust\src\etc`.

Note the fully qualified path its under and edit `C:\tools\gdb-7.9.1-tdm64-2\bin\gdbinit` to insert the path using *forward* slashes.

```
python
print "---- Loading Rust pretty-printers ----"

sys.path.insert(0, "C:/users/MyName/.rustup\toolchains/stable-x86_64-pc-windows-gnu/lib/rustlib/src/rust/src/etc")
import gdb_rust_pretty_printing
gdb_rust_pretty_printing.register_printers(gdb)

end
```

Setting up an IDE

Rust is still behind some other languages when it comes to IDE integration but there are already plugins that provide much of the functionality you need.

Popular IDEs such as Eclipse, IntelliJ, Visual Studio all have plugins that work to varying degrees of integration with Rust.

- Visual Studio Code (not to be confused with Visual Studio) is a cross-platform programming editor and has a lot of plugins. It can be set up into a complete Rust development environment by following this tutorial.
- Rust plugin for IntelliJ IDEA is under active development. This plugin has a lot of traction and is turning around new versions on a nearly weekly basis. Offers syntax highlighting, autocomplete (via built-in parser), cargo builts and eventually other functionality. IntelliJ is a commercial product but it comes in a community edition which is sufficient for development.

- Visual Rust plugin for Microsoft Studio . Offers syntax highlighting, autocompletion, interactive debugging.
- RustDT for Eclipse is also under active development. It adds syntax highlighting, autocomplete (via racer), cargo builds and rustfmt functionality to Eclipse.
- Atom is a popular editor with heaps of plugins. These plugins are very useful for Rust:
  - language-rust provides basic syntax highlighting
  - racer for autocompletion functionality
  - atom-beautify invokes rustfmt to make code look pretty.
  - build-cargo invokes cargo for you showing errors and warnings inline.

For other editors and IDEs refer to the Rust and IDEs page on the Rust website.

Racer / Rustfmt

Some of the plugins above make use of Racer and Rustfmt.

Racer is used by some plugins to provide autocompletion functionality.

Rustfmt is a source code formatting tool that makes sure your Rust source code is pretty to look at, adding spacing, indentation and so on.

You can get both just by typing these commands and waiting for the tools to download and build themselves - they're written in Rust and built through cargo.

```
cargo install racer
cargo install rustfmt
```

C and C++ Background

This section talks about C and C++. It describes its history, standards and provides a background as to how it ended up where it is today.

History of C

Early Days

The creation of C is closely associated with the early days of Unix. Bell Labs developed Unix out of an earlier project called Multics. The first version of Unix ran on PDP-7 microcomputer and funding was given to move it to PDP-11. Dennis Ritchie was a key member on this project and set about creating a language that could help him develop Unix while minimizing the amount of assembly language he had to write. Most of the code up to that point was expressed in assembly language which was error prone and obviously non portable.

Ritchie developed C so that he could write code in terms of variables, expressions, loops, functions etc. and use a *compiler* to translate C code into machine code. The generated code ran almost as fast as hand written assembly and was more portable since only the compiler had to be changed in order to support a new architecture. C itself was influenced by B (hence why it was called C), which itself was influenced by BCPL.

Defacto standard and emerging popularity

In 1978 C was formalised into a defacto standard called K&R C, named after Brian Kernighan & Dennis Ritche who published the standard as a book.

Over time the use of C became more widespread and compilers such as Turbo C, Lattice C, Microsoft C popularized C on other operating systems including personal computers.

International Standards

C later became an ANSI standard, C89. A further standard followed with C99 and C is still under review and development.

Some functionality that was introduced in C++ has also found its way back into C standards. For example, the // style single-line comment and variable declaration rules in blocks.

History of C++

C++ first appeared in 1983 as C with classes. It was invented by Bjarne Stroustrop as a way to imbue C with Simula-like features. Simula is a language that allowed concepts such as objects, classes and inheritance to be expressed in code and as its name suggests was created for running simulations. However it was considered too slow for systems programming and so something that combined speed of C with object oriented concepts was highly desirable.

C++ added these concepts as extensions to the C language and used a precompiler called `cfront` to transform the C++ extensions into C code that could then be compiled into machine code. So a C++ program could have the high level object oriented concepts but without the overhead that came with Simula.

C++ became popular in its own right and outgrew the limitations of cfront preprocessor to become supported by compilers in its own right. Thus toolchains such as Microsoft Visual C++, GCC, Clang etc. support both languages. Some toolchains have also been given to favouring C++ over C, for example Microsoft's compiler has been very slow to implement C99.

Object oriented programming has mostly been used in higher level software - applications, games, simulations and mathematical work.

C++ has also become formalised standards with C++98, C++03, C++11 and so on.

Modern C++

C++11 onwards is a distinctly different beast from earlier iterations and strives to add functionality that if used correctly can eliminate a lot of issues that will be discussed later on:

- Scoped and shared pointers
- auto keyword
- move semantics (i.e. moving data ownership of data from one variable to another)
- rvalue references
- perfect forwarding
- nullptr explicit type

However it is worth noting that since many of these things are late additions to C++. Things like move semantics must be explicitly used and have implications that are not an issue for Rust where they have been part of the language since early on.

The relationship between C and C++

While C++ grew out of C and has developed alongside it, it is not true to say C++ is a superset of C. Rather it is *mostly* a superset. There are differences such as keywords and headers that C recognizes that C++ does not.

C++ has function overloading and classes and uses name mangling to disambiguate overloaded functions. But in practice it is possible to write C as a subset of C++ and compile the two into the same executable. Most real-world C code could be called C++ *without* classes.

C and C++ are even usually handled by the same toolchain. Most compilers would consist of a front half that parses the language into an intermediate form and a back half which turns the intermediate form into optimized machine code. Finally the linker would join all the binary objects together to form an executable. C and C++ would share most of this code path.

C++ tends to be more popular with applications level programming. Part of the reason C++ hasn't found itself in the lower layers is the perception that exception handling, name mangling, linking and issues of that nature add unwanted complexity or that somehow the generated code is less efficient. Arguments have been made that this is not the case, but the perception still remains.

C still tends to be more popular in low level systems programming. Components such as the Linux kernel are pure C with some assembly. Many popular open source libraries such as sqlite3 are also written in C.

Objective-C

Objective-C is another C derived language that added objects and classes. Unlike C++, Objective-C behaves as a strict superset of C.

The language was developed in the 1980s and was popularized in the NeXTSTEP operating system and later in Apple's OS X and iOS. It hasn't gained much popularity outside of those platforms but the success of the iPhone has ensured it has a sizeable developer base of its own. It is also well supported by the GCC and Clang toolchains. Apple has begun to deprecate Objective-C in favour of Swift which is a modern high level language similar in some respects to Rust but more application focussed.

Objective-C is strongly influenced by Smalltalk (as opposed to Simula in C++) and so code works somewhat differently than C++.

Notionally code calls objects by sending them a message. An object defines an interface specifying what messages it accepts and an implementation that binds those messages to code. The caller code sends a message to call a method. Objects can also receive dynamic messages, i.e. ones not defined by their interfaces, so they can do certain tasks such as

intercepting and forwarding messages. In addition an object can ignore a message or not implement it without it being considered an error. In a broad sense, an ObjC message and a C++ method are or more or less analogous in functionality.

C/C++ Timeline

These are the major revisions of C and C++

Year	Event	Description
1972	C	C for PDP-11, other Unix systems
1978	K&R C	C as defined in "The C Programming Language" book by Kernighan & Ritchie
1989	C89 (ANSI X3.159-1989)	C is standardized as ANSI C, or C89. C90 (ISO/IEC 9899:1990) is the ISO ratified version of this same standard.
1979	C with classes -> C++	Bjarne Stroustrops
1995	C95 (ISO/IEC 9899/AMD1:1995)	Wide character support, digraphs, new macros, and some other minor changes.
1998	C++98 (ISO/IEC 14882:1998)	C++ is standardized for the first time.
1999	C99 (ISO/IEC 9899:1999)	Single line (//) comments, mixing declarations with code, new intrinsic types, inlining, new headers, variable length arrays
2003	C++03 (ISO/IEC 14882:2003)	Primarily a defect revision, addressing various defects in the specification.
2011	C++11 (ISO/IEC 14882:2011)	A major revision that introduces type inference (auto), range based loops, lambdas, strongly typed enums, a nullptr constant, struct initialization. Improved unicode char16_t, char32_t, u, U and u8 string literals.
2011	C11 (ISO/IEC 9899:2011)	Multi-threading support. Improved unicode char16_t, char32_t, u, U and u8 string literals. Other minor changes
2014	C++14 (ISO/IEC 14882:2014)	Another major revision that introduces auto return types, variable templates, digit separators (1'000'000), generic lambdas, lambda capture expressions, deprecated attribute.

Rust Background

The catalyst for Rust was the Mozilla Firefox web browser. Firefox like most web browsers is:

- Written in C++. [1]
- Complex with millions of lines of code.
- Vulnerable to bugs, vulnerabilities and exploits, many of which are attributable to the language the software is written in.
- Mostly single-threaded and therefore not well suited for many-core devices - PCs, phones, tablets etc. Implementing multi-threading to the existing engine would doubtless cause even more bugs and vulnerabilities than being single threaded.

Rust was conceived as a way to obtain C or C++ levels of performance but also remove entire classes of software problem that destabilize software and could be exploited. Code that passes the compiler phase could be guaranteed to be memory safe and therefore could be written in a way to take advantage of concurrency.

So Rust began life as a research project by Graydon Hoare in 2009 for the Mozilla foundation to solve these issues. It progressed until the release of version 1.0 in 2015.

The project is hosted on GitHub. The language has been *self-hosting* for quite some time - that is to say the Rust compiler is written in Rust, so compiling Rust happens from a compiler written in Rust. Get your head around that! But it's the same way that C and C++ compilers are these days too.

[1] Read this Mozilla internal string guide to get a flavor of the sort of problems the browser had to overcome. A browser obviously uses a lot of temporary strings. STL strings were too inefficient / flakey from one compiler to the next and so the browser sprouted an entire tree of string related classes to solve this issue. Similar tales are told in Qt and other large libraries.

Problems with C/C++

It is trivial (by accident) to write code that is in error such as causing a memory leak. It is easy (by malice) to exploit badly written code to force it into error. It easy with the best testing in the world for some of these errors to only manifest themselves when the code is in production.

At best, bugs are a costly burden for developers to find and fix, not just in time and dollars but also their reputation. At worst, the bug could causes catastrophic failure but more ordinarily leaves code unstable or vulnerable to hacking.

Rust is a language that produces machine code that is comparable in performance as C/C++ but enforces a safe-by-design philosophy. Simply put, the language and the compiler try to stop errors from happening in the first place. For example the compiler rigorously enforces lifetime tracking on objects and generates errors on violations. Most of these checks and guards are done at compile time so there is a zero-cost at runtime.

Active Development

The Rust team releases a new version of Rust approximately every 6 weeks. This means Rust receives code and speed improvements over time.

Most releases focus on marking APIs as stable, improving code optimization and compile times.

Open source and free

Rust is dual licensed under the Apache 2.0 and MIT open source licenses. The full copyright message is viewable online.

Essentially the license covers your right to modify and distribute the Rust source code. Note that Rust generates code for LLVM so LLVM also has its own software license (TODO link).

What you compile with Rust (or LLVM) is not affected by the open source license. So you may compile, execute and distribute proprietary code without obligation to these licenses.

Is Rust for everybody?

No of course not. Performance and safety are only two things to consider when writing software.

- Sometimes it's okay for a program to crash every so often
- If you have code that's written and works then why throw that away?
- Writing new code will always take effort and will still cause application level bugs of one sort or another.
- Performance may not be a big deal especially for network bound code and a higher level language like Java, C#, Go may suit better.

- Some people will find the learning curve extremely steep.
- Rust is still relatively immature as a language and still has some rough edges - compilation times, optimization, complex macros.

But you may still find there is benefit to moving some of your code to Rust. For example, your C++ software might work great but it has to deal with a lot of user-generated data so perhaps you want to reimplement that code path in Rust for extra safety.

Safe by design

Some examples of this safe-by-design philosophy:

- Variable (binding) is immutable by default. This is the opposite of C++ where mutable is the default and we must explicitly say const to make something immutable. Immutability extends to the &self reference on struct functions.
- Lifetime tracking. The Rust compiler will track the lifetime of objects and can generate code to automatically drop them when they become unused. It will generate errors if lifetime rules are violated.
- Borrowing / Variable binding. Rust enforces which variable "owns" an object at any given time, and tracks values that are moved to other variables. It enforces rules about who may hold a mutable or immutable reference to it. It will generate errors if the code tries to use moved variables, or obtain multiple mutable references to it.
- There is no NULL pointer in safe code. All references and pointers are valid because their lifetimes and borrowing are tracked.
- Rust uses LLVM for the backend so it generates optimized machine code.
- Lint checking is builtin, e.g. style enforcement for naming conventions and code consistency.
- Unit tests can be integrated into the code and run automatically
- Modules (equivalent to namespaces C++) are automatic meaning we implicitly get them by virtue of our file structure.

Don't C++11 / C++14 get us this?

Yes and no. C++11 and C++14 certainly bring in some long overdue changes. Concurrency primitives (threads at last!), move semantics, pointer ownership and other beneficial things all come in with these latest standards. Conveniences such as type inference, lambdas et al also come in.

And perhaps if you program the right subset of features and diligently work to avoid pitfalls of C++ in general then you are more likely to create safe code.

But what is the *right* subset?

- If you use someone else's library - are they using the right subset?
- If one subset is right then why does C++ still contain all the stuff that is outside of that?
- Why are all the things which are patently unsafe / dangerous still allowed?
- Why are certain dangerous default behaviors such as default copy constructors not flipped to improve code safety?

We could argue that C++ doesn't want to break existing code by introducing change that requires code to be modified. That's fair enough but the flip-side is that future code is almost certainly going to be broken by this decision. Perhaps it would be better to inflict a little pain for some long term gain.

Unsafe programming / C interoperability

Rust recognizes you may need to call an external libraries, e.g. in a C library or a system API.

Therefore it provides an `unsafe` keyword that throws some of the safety switches when it is necessary to talk to the outside world.

This allows you consider the possibility of porting code partially to Rust while still allowing some of it to remain as C.

Let's Start Simple

The usual introduction to any language is "Hello, World!". A simple program that prints that message out to the console.

Here is how we might write it for C:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

C++ could write it the same way, or we could use the C++ stream classes if we preferred:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
  cout << "Hello, World!" << endl;
  return 0;
}
```

And here is the equivalent in Rust:

```
fn main() {
  println!("Hello, World!");
}
```

There are some obvious points of similarity that we can observe:

- C/C++ and Rust follow the convention of having a `main()` function as the entry point into code. Note that Rust's main doesn't return anything. It's effectively a void method.
- There is a general purpose print statement.
- The general structure in terms of main, use of { } and semi-colons is mostly the same. In both languages a block of code is enclosed in curly braces, and a semi-colon is used as

a separator between statements.
- Rust looks a little bit more terse than either C or C++ because it automatically includes references to part of its standard runtime that it refers to as its "prelude".

The `println!()` is actually a macro that expands into code that writes to the standard output. We know it's a macro because it ends in a ! character but you may treat it like a function call for now. We'll see how Rust macros differ to those in C/C++ later.

Compiling our code

Open a command prompt and set up your compiler environments.

If you were using gcc, you'd compile your code like this:

```
gcc hw.cpp -o hw
```

If you were using Microsoft Visual C++ you'd compile like this:

```
cl /o hw.exe hw.cpp
```

To compile in Rust you invoke the rustc compiler.

```
rustc hw.rs
```

And to run either

```
./hw (or .\hw.exe)
Hello, World!
```

Again there are points of similarity:

- There is a shell command that compiles the code and creates an executable from it.
- The binary runs in the same way.

A less obvious point of similarity is that Rust shares its code generation backend with gcc-llvm and clang. Rustc outputs llvm bitcode which is compiled (and optimized) into machine code via LLVM. This means the resulting executable is very similar in form to that output by C++ compilers. That includes the symbolic information it supplies for debugging purposes. A rust executable can be debugged in gdb, lldb or Microsoft Visual Studio depending on the target platform.

```
rustc -O hw.rs
```

Compiling and Linking in More Detail

Your main() entry point

Rust has a main function just like C/C++ which is usually called `main()`. [1]

It doesn’t take any arguments and it doesn’t return anything unlike C/C++. Let's see how we might do those things.

Processing command-line arguments

In C/C++, the entry point takes argc, and argv arguments. Argc is the number of arguments and argv is an array of char * pointers that specify those arguments.

```
int main(int arcg, char **argv) {
  // our code
}
```

Processing arguments can become inordinately complex (and buggy) so most software will use a function like `getopt()` or `getopt_long()` to simplify the process.

Note that `getopt()` is not a standard C function and is not portable, e.g. to Windows. So immediately we see an example of problem that C/C++ forces us to solve.

Rust doesn't process arguments this way. Instead you access the command-line parameters from `std::env::args()` from anywhere in the code. That is to say, there is a function called `args()` under the namespace `std::env` that returns the strings on the command-line.

The function `args()` returns the parameters in a string array. As with C++, the first element of the array at index 0 is the command itself:

```
use std::env;
fn main() {
    for argument in env::args() {
        println!("{}", argument);
    }
}
```

Alternatively, since `args()` returns a type called `Args` that implements the `Iterator` trait you can collect the arguments up into your own collection and process that:

```
use std::env;
use std::collections::HashSet;
fn main() {
    let args: HashSet<String> = env::args().collect();
    let verbose_flag = args.contains("--verbose");
}
```

We can see some clear advantages to how Rust supplies args:

- You don't need a separate argc, parameter. You have an array that defines its own length.
- You can access arguments from anywhere in your program, not just from the `main()`. In C++ you would have to pass your args around from one place to another. In Rust you can simply ask for them from anywhere.

Use a crate - easy command-line processing

Rust has a number of crates for processing arguments. The most popular crate for processing arguments is clap.

It provides a very descriptive, declarative way of adding rules for processing arguments into the code. It is especially useful if your program takes a lot of arguments, including parameters and validation rules.

For example we add this to `Cargo.toml`:

```
[dependencies]
clap = "2.27"
```

And in our `main.rs`.

```
#[macro_use] extern crate clap;
use clap::*;
fn main() {
  let matches = App::new("Sample App")
    .author("My Name <myname@foocorp.com>")
    .about("Sample application")
    .arg(Arg::with_name("T")
        .long("timetowait")
        .help("Waits some period of time for something to
happen")
        .default_value("10")
        .takes_value(true)
        .possible_values(&["10", "20", "30"])
        .required(false))
    .get_matches();

  let time_to_wait = value_t_or_exit!(matches, "T", u32);
  println!("Time to wait value is {}", time_to_wait);
}
```

This code will process arguments for `-T` or `--timetowait` and ensure the value is one of 3 accepted. And if the user doesn't supply a value, it defaults to `10`. And if the user doesn't supply a valid integer it will terminate the application with a useful error.

The user can also provide `--help` as an argument and it will print out the usage.

Exit code

If you want to exit with a code, you set it explicitly:

```
fn main() {
    //... my code
    std::os::set_exit_status(1);
}
```

When `main()` drops out, the runtime cleans up and returns the code to the environment. Again there is no reason the status code has to be set in `main()`, you could set it somewhere else and `panic!()` to cause the application to exit.

Optimized compilation

In a typical edit / compile / debug cycle there is no need to optimize code and so Rust doesn't optimize unless you ask it to.

Optimization takes longer to happen and can reorder the code so that backtraces and debugging may not point at the proper lines of code in the source.

If you want to optimize your code, add a -O argument to rustc:

```
rustc -O hw.rs
```

The act of optimization will cause Rust to invoke the LLVM optimizer prior to linking. This will produce faster executable code at the expense of compile time.

Incremental compilation

Incremental compilation is also important for edit / compile / debug cycles. Incremental compilation only rebuilds those parts of the code which have changed through modification to minimize the amount of time it takes to rebuild the product.

Rust has a different incremental compilation model to C++.

- C++ doesn't support incremental compilation per se. That function is left to the make / project / solution tool. Most builders will track a list of project files and which file depends on other files. So if file foo.h changes then the builder knows what other files depend on it and ensures they are rebuilt before relinking the target executable.
- In Rust incremental compilation is at the crate level - that if any file in a crate changes then the crate as a whole has to be rebuilt. Thus larger code bases tend to be split up into crates to reduce the incremental build time.

There is a recognition in the Rust community that the crate-level model can suck for large crates so the Rust compiler is getting incremental per-file compilation support in addition to per-crate.

At the time of writing this support is experimental because it is tied to refactoring the compiler for other reasons to improve performance and optimization but will eventually be enabled and supported by rustc and cargo.

Managing a project

In C++ we would use a `makefile` or a solution file of some kind to manage a real world project and build it.

For small programs we might run a script or invoke a compiler directly but as our program grows and takes longer to build, we would have to use a `makefile` to maintain our sanity.

A typical `makefile` has rules that say what files are our sources, how each source depends on other sources (like headers), what our final executable is and a bunch of other mess about compile and link flags that must be maintained.

There are lots of different makefile solutions which have cropped up over the years but a simple gmake might look like one:

```
SRCS = main.o pacman.o sprites.o sfx.o
OBJS = $(SRCS:.cpp=.o)
EXE = pacman
$(EXE): $(OBJS)
    $(CC) $(CFLAGS) -o $(EXE) $(OBJS)
.cpp.o:
    $(CC) $(CFLAGS) -c $< -o $@
```

When you invoke `make`, the software will check all the dependencies of your target, looking at their filestamps and determine which rules need to be invoked and which order to rebuild your code.

Rust makes things a lot easier – there is no makefile! The source code is the makefile. Each file says what other files it uses via depencies on other crates, and on other modules.

Consider this main.rs for a pacman game:

```
mod pacman;

fn main() {
  let mut game = pacman::Game::new();
  game.start();
}
```

If we save this file and type `rustc main.rs` the compiler will notice the reference to `mod pacman` and will search for a `pacman.rs` (or `pacman/mod.rs`) and compile that too. It will continue doing this with any other modules referenced along the way.

In other words you could have a project with 1000 files and compile it as simply as `rustc main.rs`. Anything referenced is automatically compiled and linked.

Okay, so we can call `rustc`, but what happens if our code has dependencies on other projects. Or if our project is meant to be exported so other projects can use it?

Cargo

Cargo is a package manager build tool rolled into one. Cargo can fetch dependencies, build them, build and link your code, run unit tests, install binaries, produce documentation and upload versions of your project to a repository.

The easiest way to create a new project in Rust is to use the `cargo` command to do it

```
cargo new hello_world –bin
```

Creates this

```
hello_world/
  .git/ (git repo)
  .gitignore
  Cargo.toml
  src/
    main.rs
```

Building the project is then simply a matter of this:

```
cargo build
```

If you want to build for release you add a --release argument. This will invokes the rust compiler with optimizations enabled:

```
cargo build --release
```

If we wanted to build and run unit tests in our code we could write

```
cargo test
```

Crates and external dependencies

Cargo doesn't just take care of building our code, it also ensures that anything our code depends on is also downloaded and built. These external dependencies are defined in a `Cargo.toml` in our project root.

We can edit that file to say we have a dependency on an external "crate" such as the `time` crate:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Joe Blogs <jbloggs@somewhere.com>"]

[dependencies]
time = "0.1.35"
```

Now when we run `cargo build`, it will fetch "time" from crates.io and also any dependencies that "time" has itself. Then it will build each crate in turn automatically. It does this efficiently so iterative builds do not incur a penalty. External crates are download and built in your .cargo home directory.

To use our external crate we declare it in the main.rs of our code, e.g.

```
extern crate time;

fn main() {
  let now = time::PreciseTime::now();
  println!("The time is {:?}", now);
}
```

So the change to the `Cargo.toml` and a reference in the source is sufficient to:

1. Fetch the crate (and any dependencies)
2. Build the crate (and any dependencies)
3. Compile and link to the crate and dependencies

All that happened with a line in `Cargo.toml` and a line in our code to reference the crate. We didn't have to mess around figuring how to build the other library, or maintain multiple makefiles, or getting our compiler / linker flags right. It just happened.

Cargo.lock

Also note that once we build, cargo creates a `Cargo.lock` file in our root directory.

This file is made so that if `cargo build` is invoked again it has an exact list of what packages need to be pulled and compiled. It stops situations where the code under our feet (so to speak) moves and suddenly our project no longer builds. So if the lock file exists, the same dependency configuration can be reproduced even from a clean. If you want to force the cargo to rebuild a new lock file, e.g. after changing `Cargo.toml`, you can type `cargo update`.

> [1]. You can change the main entry point using a special `#[start]` directive if you want on another function but the default is main() ↩

Source Layout and Other General Points

Header files

C/ C++

C and C++ code tends to be split over two general kinds of file:

- The Header file (.h, .hpp) contains class definitions, external function signatures, macros, templates, inline functions. Sometimes inline functions get stored in their own file. The standard template library C++ headers do not have a file extension. Some 3rd party libraries like Qt may sometimes omit the extension.
- The Source file (.c, .cc, .cpp) contains the implementation of classes and anything private. Sometimes C++ will use tricks such as forward class references and Pimpl patterns to keep complex or dependent code out of the header file.

Occasionally you may also see files with a .inl, or .ipp extension which are headers with a lot of inline templates or functions.

Compilers are only interested in source files and what they `#include` so what's really happening in most C/C++ code is that a preprocessor concatenates various header files to the front of the source file according to the `#` directives within it and the resulting file is fed to a compiler.

Splitting definition and implementation across multiple files can be a nuisance since it means that changes to a single class can require modifications to multiple files.

Rust

Rust does not have header files. Every struct, implementation and macro resides in a file ending in .rs. Code is made public or not by structuring .rs files into modules and exposing functions via the `pub` keyword.

Ordering is less important too. It is possible to forward reference structs or functions, or even `use` the very same module that a piece of code is a part of. The only time that ordering matters is for macro definitions. A macro must be defined before a module that uses it.

Rust files reference non-dependent modules with the `use` keyword and pull-in dependent modules with the `mod` keyword.

Namespaces

C / C++

C does not use namespaces. Libraries tend to prefix their functions and structs with a qualifying name of some sort.

C++ *does* have namespaces but their use is optional and varies from one piece of code to the next.

Rust

Rust has modules which are like `#include` and namespaces rolled into one

One major convenience definition and implementation are one and the same. Implementing a function brings it into existence. Any other module that chooses to "use" it simply says so and the compiler will ensure it compiles properly.

See Namespacing with modules TODO ref

File name conventions

In C++ filenames typically end in:

- .h, .hpp, .inl for headers or inline code
- .c, .cpp, .cc for source code

Aside from the extension (which may kick off the compiler expecting C or C++) there is next to no expected arrangement or naming convention for files.

You can compile a file called deeply/nested/Timbuktu.cpp which defines 20 classes and 30 interfaces if you like and the name does not matter.

Rust files are snake_case and end in .rs. The filename DOES matter because the name is the module name that scopes whatever is in it. There are also some special files called main.rs, lib.rs and mod.rs.

So if you name your file foo.rs, then everything inside is scoped foo::* when externally referenced.

Unicode support

Using Unicode in C++ has always been a pain. Neither C nor C++ had support for it at all, and various solutions have appeared over time. Recent implementations of the standards of C and C++ provide string literal types for UTF encodings, but prior to that it was strictly ascii or wide characters.

Here are some general guidelines for Unicode in C / C++:

- Source code is normally only safe to use characters 0-127 although some compilers may have parameters that allow makefiles to specify other encodings.
- C++ has char and wchar_t types for 8-bit and 32-bit or possibly 16-bit wide strings. Part of the problem with wchar_t was the width was immediately subverted.
- Char type implies no encoding. It normally means ASCII but could also mean UTF-8, Latin1, or in fact any form of encoding that predates Unicode. Basically it is up to the program to understand the meaning.
- A "wide" wchar_t is NOT UTF-32. It might be, or it might be UTF-16 on some platforms (e.g Windows). This messed up definition makes operations such as slicing strings dangerous due to the risk of cutting through a control point.
- What if I want to read Unicode arguments from the command-line such as file paths - what encoding are they in? The main() method passes them as char*. Windows has a wmain() that takes wchar_t*. What am I supposed to do?
- Windows favours wide (UTF-16) character strings for its APIs although it has ASCII versions too. The ASCII versions are not UTF-8. Compiled code has #define UNICODE to support multiple languages.
- Linux tends to favour UTF-8 encoded char strings. Most languages, toolkits and tools assume UTF-8. The exception is Qt which has chosen to use UTF-16 internally.
- C-lib has acquired various wide versions of its strdup, strcmp, strcpy etc. It also acquired wide versions of functions for opening files on disk and so forth.
- C++ lib has acquired std::string / std::wstring classes. C++ has acquired explicit UTF-16 and UTF-32 versions of these classes.
- C11 and C++11 introduce explicit string literals for various UTF widths.
- Limited conversion capabilities between wide / narrow in C++. Some operating systems have incomplete conversion capabilities.
- 3rd party conversion libraries like ICU4C are commonly used. Libraries like boost, Qt use libicu for converting between encodings
- Embedding Unicode into C source involves using escape codes or hex values

Rust simplifies things a lot by benefit of hindsight.

- Source code is UTF-8 encoded.
- Comments, characters and string literals can contain Unicode characters without escaping.
- The native char type is 4 bytes wide – as wide as a Unicode characters.

- The native str & String types are internally UTF-8 to save space but may be iterated by char or by byte according to what the function is doing.

Since source code is UTF-8 encoded you may embed strings straight into the source.

```
let hello = "你好";
for c in hello.chars() { /* iterate chars */
  //...
}
```

Namespacing With Modules

C++ namespaces allow you to group your functions, variables and classes into logical blocks and allow the compiler to disambiguate them from other functions, variables and classes that might otherwise have the same name.

```
// Namespacing is usually a good idea
namespace myapp {
  void error() {
    //...
  }
  const int SOME_VALUE = 20;
  void doSomething(int value) {
    //...
  }
}
//... somewhere else in the code
myapp::doSomething(100);
```

Namespacing in C++ is completely optional which means some code may use nest namespaces while other code may be content to cover its entire codebase with a single namespace. Some code might even put its code into the global namespace. Other code might control the use of namespaces with macros.

The equivalent to a namespace in Rust is a module and serves a similar purpose. Unlike C++ though you get namespacing automatically from the structure of your files. Each file is a module in its own right.

So if we may have a file myapp.rs

```
// myapp.rs
pub fn error() { /* ... */ }
pub const SOME_VALUE: i32 = 20;
pub fn doSomething(value: i32) { /* ... */ }
```

Everything in myapp.rs is automatically a module called myapp. That means modules are implicit and you don't have to do anything except name your file something meaningful.

```
use myapp;
myapp::doSomething(myapp::SOME_VALUE);
```

You could also just bring in the whole of the mod if you like:

```
use myapp::*;
doSomething(SOME_VALUE);
```

Or just the types and functions within it that you use:

```
use myapp::{doSomething, SOME_VALUE}
doSomething(SOME_VALUE);
// Other bits can still be referenced by their qualifying mod
myapp::error();
```

But if you want an explicit module you may also write it in the code. So perhaps myapp doesn't justify being a separate file.

```
// main.rs
mod myapp {
  pub fn error() { /* ... */ }
  pub const SOME_VALUE = 20;
  pub fn doSomething(value: i32) { /* ... */ }
}
```

Modules can be nested so a combination of implicit modules (from file names) and explicit modules can be used together.

Splitting modules across files

Namespacing with modules is pretty easy, But sometimes you might have lots of files in a module and you don't want the outside world to see a single module namespace.

In these cases you're more likely to use the myapp/mod.rs form. In this instance the mod.rs file may pull

in subordinate files

```
// myapp/mod.rs
mod helpers;
mod gui;

#[cfg(test)]
mod tests

// Perhaps we want the outside world to see myapp::Helper
pub use helpers::Helper;
```

In this example, the module pulls in submodules `helpers` and `gui`. Neither is marked as `pub mod` so they are private to the module.

However the module also says `pub use helpers::Helper` which allows the outside to reference `myapp::Helper`. Thus a module can act as a gatekeeper to the things it references, keeping them private or selectively making parts public.

We haven't mentioned the other module here `tests`. The attribute `#[cfg(test)]` indicates it is only pulled in when a unit test executable is being built. The `cfg` attribute is used for conditional compliation.

Using a module

Modules can be used once they are defined.

```
use helpers::*;
```

Note that the use command is relative to the toplevel `main` or `lib` module. So if you declare a `mod helpers` at the top, then the corresponding `use helpers` will retrieve it. You can also use relative `use` commands with the `super` and `self` keywords.

// TODOs

Module aliasing

TODO

External crates

TODO

Porting Code

Before starting, the assumption at this point is you *need* to port code. If you're not sure you do to port code, then maybe you don't. After all, if your C/C++ code works fine, then why change it?

TODO This section will provide a more real world C/C++ example and port it to the equivalent Rust

Automation tools

Corrode

Corrode is a command-line tool that can partially convert C into Rust. At the very least it may spare you some drudgery ensuring the functionality is as close to the original as possible.

Corrode will take a C file, e.g. `somefile.c` plus any arguments from `gcc` and produces a `somefile.rs` which is the equivalent code in Rust.

It works by parsing the C code into an abstract syntax tree and then generating Rust from that.

Interestingly Corrode is written in Haskell and more interestingly is written as a literate Haskell source - the code is a markdown document interspersed with Haskell.

Bindgen

Bindgen is a tool for generating FFI interfaces for Rust from existing C and C++ header files. You might find this beneficial if you're porting code from C / C++, or writing a new component that must work with an existing code base.

Bindgen requires that you preinstall the Clang C++ compiler in order to parse code into a structure it can digest.

The readme documentation on the site link provides more information on installing and using the tool.

Experiences

A number of websites offer insights of the porting process from C to Rust

1. Porting Zopfli from C to Rust. Zopfli is a library that performs good but slow deflate algorithm. It produces smaller compressed files than zlib while still remaining compatible with it.
2. TODO

Tips

Use references wherever you can

TODO references are equivalent to C++ references or to pointers in C. Passing by reference is an efficient way of passing any object greater than 64-bits in size into a function without relinquishing ownership. In some cases, it is even a good idea to return by reference since the caller can always clone the object if they need to, and more often than not they can just use the reference providing the lifetimes allow for it.

TODO you do not need to use references on intrinstic types such as integers, bools etc. and there is no point unless you intend for them to be mutable and change.

Learn move semantics

C and C++ default to copy on assign, Rust moves on assign unless the type implements the `Copy` trait. This is easily one of the most mind boggling things that new Rust programmers will encounter. Code that works perfectly well in C++ will instantly fail in Rust.

The way to overcome this is first use references and secondly don't move unless you intend for the recipient to be the new owner of an object.

TODO if you intend for the recipient to own a copy of an object then implement the Clone trait on your struct. Then you may call `.clone()` and the recipient becomes the owner of the clone instead of your copy.

Use modules to naturally arrange your source code

TODO

Using composition and traits

TODO Rust does not allow you to inherit one struct from another. The manner of overcoming this.

Using Cargo.toml to create your build profiles

Use Rust naming conventions and formatting

C and C++ have never had

Foreign Function Interface

TODO for now read the FFI omnibus.

Leaving function names unmangled

TODO attribute no_mangle

libc

TODO Rust provides a crate with bindings for C library functions. If you find yourself receiving a pointer allocated with malloc you could free it with the corresponding call to free() via the bindings.

TODO add the following to your `Cargo.toml`

```
[dependencies]
libc = "*"
```

TODO example of using libc

Features of Rust compared with C++

Rust and C++ have roughly analogous functionality although they often go about it in different ways.

Rust benefits from learning what works in C / C++ and what doesn't and indeed has cherry-picked features from a variety of languages. It also enjoys a cleaner API in part because things like Unicode dictate the design.

This section will cover such topics as types, strings, variables, literals, collections, structs, loops and so on. In each case it will draw comparison between how things are in C/C++ and how they are in Rust.

Also bear in mind that Rust compiles to binary code and is *designed* to use C binaries and be used by C binaries. Therefore the generated code is similar, but it is different as source.

Types

C/C++ compilers implement a *data model* that affects what width the standard types are. The general rule is that:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

As you can see, potentially everything all the way to `long long` could be a single byte, or there could be some other crazy definition. In practice however, data models come in four common types which will be covered in the next section.

For this section, we'll cover the *most likely* analogous types between Rust and C/C++.

C/C++	Rust	Notes
`char`	`i8` (or `u8`)	The signedness of a C++ char can be signed or unsigned - the assumption here is signed but it varies by target system. A Rust `char` is not the same as a C/C++ `char` since it can hold any Unicode character. [1]
`unsigned char`	`u8`	
`signed char`	`i8`	
`short int`	`i16`	
`unsigned short int`	`u16`	
`(signed) int`	`i32` or `i16`	In C/C++ this is data model dependent [2]
`unsigned int`	`u32` or `u16`	In C/C++ this is data model dependent [2]
`(signed) long int`	`i32` or `i64`	In C/C++ this is data model dependent [2]
`unsigned long int`	`u32` or `u64`	In C/C++ this is data model dependent [2]
`(signed) long long int`	`i64`	
`unsigned long long int`	`u64`	
`size_t`	`usize`	usize holds numbers as large as the address space [3]
`float`	`f32`	
`double`	`f64`	
`long double`	~~f128~~	f128 support was present in Rust but removed due to issues for some platforms in implementing it.
`bool`	`bool`	
`void`	`()`	The unit type (see below)

[1] Rust's `char` type, is 4 bytes wide, enough to hold any Unicode character. This is equivalent to the belated `char32_t` that appears in C++11 to rectify the abused C++98 `wchar_t` type which on operating systems such as Windows is only 2 bytes wide. When you iterate strings in Rust you may do so either by character or `u8` , i.e. a byte.

[2] See the next section to for a discussion on data models.

[3]

[3] Rust has a specific numeric type for indexing on arrays and collections called `usize`. A `usize` is designed to be able to reference as many elements in an array as there is addressable memory. i.e. if memory is 64-bit addressable then usize is 64-bits in length. There is also a signed `isize` which is less used but also available.

Data model

The four common data models in C++ are:

- LP32 - `int` is 16-bit, `long` and pointers are 32-bit. This is an uncommon model, a throw-back to DOS / Windows 3.1
- ILP32 - `int`, `long` and pointers are 32-bit. Used by Win32, Linux, OS X
- LLP64 - `int` and `long` are 32-bit, `long long` and pointers are 64-bit. Used by Win64
- LP64 - `int` is 32-bit, `long` / `long long` and pointers are 64-bit. Used by Linux, OS X

C/C++ types compared to Rust

C/C++ and Rust will share the same machine types for each corresponding language type and the same compiler / backend technology, i.e.:

1. Signed types are two's complement
2. IEE 754-2008 binary32 and binary64 floating points for float and double precision types.

stdint.h / cstdint

C provides a `<stdint.h>` header that provides unambigious typedefs with length and signedess, e.g. `uint32_t`. The equivalent in C++ is `<cstdlib>`.

If you use the types defined in this header file the types become directly analogous and unambiguous between C/C++ and Rust.

C/C++	Rust
`int8_t`	`i8`
`uint8_t`	`u8`
`int16_t`	`i16`
`uint16_t`	`u16`
`uint32_t`	`u32`
`int32_t`	`i32`
`int64_t`	`i64`
`uint64_t`	`u64`

Integer types

C++

C/C++ has primitive types for numeric values, floating point values and booleans. Strings will be dealt in a separate section.

Integer types (`char` , `short` , `int` , `long`) come in `signed` and `unsigned` versions.

A `char` is always 8-bits, but for historical reasons, the standards only guarantee the other types are "at least" a certain number of bits. So an `int` is ordinarily 32-bits but the standard only say it should be at *least as large* as a `short` , so potentially it could be 16-bits!

More recent versions of C and C++ provide a `<cstdint>` (or `<stdint.h>` for C) with typedefs that are unambiguous about their precision.

Even though `<stdint.h>` can clear up the ambiguities, code frequently sacrifices correctness for terseness. It is not unusual to see an `int` used as a temporary incremental value in a loop:

```
string s = read_file();
for (int i = 0; i < s.size(); ++i) {
  //...
}
```

While `int` is unlikely to fail for most loops in a modern compiler supporting ILP32 or greater, it is still technically wrong. In a LP32 data model incrementing 32767 by one would become -32768 so this loop would never terminate if `s.size()` was a value greater than that.

But look again at this snippet. What if the file read by `read_file()` is outside of our control. What if someone deliberately or accidentally feeds us a file so large that our loop will fail trying to iterate over it? In doing so our code is hopelessly broken.

This loop should be using the same type returned from `string::size()` which is an opaque unsigned integer type called `size_type`. This is usually a typedef for `std::size_t` but not necessarily. Thus we have a type mismatch. A `string` has an iterator which could be used instead but perhaps you need the index for some reason, but it can messy:

```
string s = read_file();
for (string::iterator i = s.begin(); i != s.end(); ++i) {
  string::difference_type idx = std::distance(s.begin(), i);
  //...
}
```

Now we've swapped from one opaque type `size_type` to another called `difference_type`. Ugh.

C/C++ types can also be needlessly wordy such as `unsigned long long int`. Again, this sort of puffery encourages code to make bad assumptions, use a less wordy type, or bloat the code with typedefs.

Rust

Rust benefits from integer types that unambiguously denote their signedness and width in their name - `i16`, `u8` etc.

They are also extremely terse making it easy to declare and use them. For example a `u32` is an unsigned 32-bit integer. An `i64` is a signed 64-bit integer.

Types may be inferred or explicitly prefixed to the value:

```
let v1 = 1000;
let v2 : u32 = 25;
let v3 = 126i8;
```

Rust also has two types called `usize` and `isize` respectively. These are equivalent to `size_t` in that they are as large enough to hold as many elements as there is addressable memory. So in a 32-bit operating system they will be 32-bits in size, in a 64-bit operating system they will be 64-bits in size.

Rust will not implicitly coerce an integer from one size to another without explicit use of the `as` keyword.

```
let v1 = 1000u32;
let v2: u16 = v1 as u16;
```

Real types

C++

C/C++ has float, double and long double precision floating point types and they suffer the same vagueness as integer types.

- `float`
- `double` - "at least as much precision as a `float` "
- `long double` - "at least as much precision as a `double` "

In most compilers and architectures however a float is a 32-bit single precision value, and a double is an 64-bit double precision value. The most common machine representation is the IEEE 754-2008 format.

Long double

The `long double` has proven quite problematic for compilers. Despite expectations that it is a quadruple precision value it usually isn't. Some compilers such as gcc may offer 80-bit extended precision on x86 processors with a floating point unit but it is implementation defined behaviour.

The Microsoft Visual C++ compiler treats it with the same precision as a `double` . Other architectures may treat it as quadruple precision. The fundamental problem with `long double` is that most desktop processors do not have the ability in hardware to perform 128-bit floating point operations so a compiler must either implement it in software or not bother.

Math functions

The `<math.h>` C header provides math functions for working with different precision types.

```
#include <math.h>

const double PI = 3.1415927;
double result = cos(45.0 * PI / 180.0);
//..
double result2 = abs(-124.77);
//..
float result3 = sqrtf(9.0f);
//
long double result4 = powl(9,10);
```

Note how different calls are required according to the precision, e.g. sinf, sin or sinl. C99 supplies a "type-generic" set of macros in `<tgmath.h>` which allows sin to be used regardless of type.

C++11 provides a `<cmath>` that uses specialised inline functions for the same purpose:

```
#include <cmath>
float result = std::sqrt(9.0f);
```

Rust

Rust implements two floating point types - `f32` and `f64`. These would be analogous to a 32-bit `float` and 64-bit `double` in C/C++.

```
let v1 = 10.0;
let v2 = 99.99f32;
let v3 = -10e4f64;
```

Unlike in C/C++, the math functions are directly bound to the type itself providing you properly qualify the type.

```
let result = 10.0f32.sqrt();
//
let degrees = 45.0f64;
let result2 = angle.to_radians().cos();
```

Rust does not have a 128-bit double. A `f128` did exist for a period of time but was removed to portability, complexity and maintenance issues. Note how `long double` is treated (or not) according to the compiler and target platform.

At some point Rust might get a f128 or f80 but at this time does not have such a type.

Booleans

A `bool` (boolean) type in C/C++ can have the value `true` or `false`, however it can be promoted to an integer type (0 == `false`, 1 == `true`) and a bool even has a ++ operator for turning false to true although it has no -- operator!?

But inverting true with a ! becomes false and vice versa.

```
!false == true
!true == false
```

Rust also has a `bool` type that can have the value `true` or `false`. Unlike C/C++ it is a true type with no promotion to integer type

void / Unit type

C/C++ uses `void` to specify a type of nothing or an indeterminate pointer to something.

```
// A function that doesn't return anything
void delete_directory(const std::string &path);

// Indeterminate pointer use
struct file_stat {
  uint32_t creation_date;
  uint32_t last_modified;
  char file_name[MAX_PATH + 1];
};

// malloc returns a void * which must be cast to the type need
file_stat *s = (file_stat *) malloc(sizeof(file_stat));
// But casting is not required when going back to void *
free(s);
```

The nearest thing to `void` in Rust is the Unit type. It's called a Unit type because it's type is `()` and it has one value of `()`.

Technically `void` is absolutely nothing and `()` is a single value of type `()` so they're not analogous but they serve a similar purpose.

When a block evaluates to nothing it returns `()`. We can also use it in places where we don't care about one parameter. e.g. say we have a function `do_action()` that succeeds or fails for various reasons. We don't need any payload with the Ok response so specify `()` as the payload of success:

```
fn do_action() -> Result<(), String> {
 //...
 Result::Ok(())
}

let result = do_action();
if result.is_ok() {
 println!("Success!");
}
```

Empty enums

Rust *does* have something closer (but not the same as) `void` - empty enumerations.

```
enum Void {}
```

Essentially this enum has no values at all so anything that assigns or matches this nothing-ness is unreachable and the compiler can issue warnings or errors. If the code had used `()` the compiler might not be able to determine this.

Tuples

A tuple is a collection of values of the same or different type passed to a function or returned by one as if it were a single value.

C/C++ has no concept of a tuple primitive type, however C++11 can construct a tuple using a template:

```
std::tuple<std::string, int> v1 = std::make_tuple("Sally", 25);
//
std::cout << "Name = " << std::get<0>(v1)
          << ", age = " << std::get<1>(v1) << std::endl;
```

Rust supports tuples as part of its language:

```
let v1 = ("Sally", 25);
println!("Name = {}, age = {}", v1.0, v1.1);
```

As you can see this is more terse and more useful. Note that the way a tuple is indexed is different from an array though, values are indexed via .0, .1 etc.

Tuples can also be returned by functions and assignment operators can ignore tuple members we're not interested in.

```
let (x, y, _) = calculate_coords();
println!("x = {}, y = {}", x, y);
//...
pub fn calculate_coords() -> (i16, i16, i16) {
  (11, 200, -33)
}
```

In this example, the calculate_coords() function returns a tuple containing three `i16` values. We assign the first two values to `x` and `y` respectively and ignore the third by passing an underscore. The underscore tells the compiler we're aware of the 3rd value but we just don't care about it.

Tuples can be particularly useful with code blocks. For example, let's say we want to get some values from a piece of code that uses a guard lock on a reference counted service. We can lock the service in the block and return all the values as a tuple to the recipients outside of the block:

```
let protected_service: Arc<Mutex<ProtectedService>> =
Arc::new(Mutex::new(ProtectedService::new()));
//...
let (host, port, url) = {
  // Lock and acquire access to ProtectedService
  let protected_service = protected_service.lock().unwrap();
  let host = protected_service.host();
  let port = protected_service.port();
  let url = protected_service.url();
  (host, port, url)
}
```

This code is really neat - the lock allows us to obtain the values, the lock goes out of scope and the values are returned in one go.

Arrays

An array is a fixed size list of elements allocated either on the stack or the heap.

E.g to create a 100 element array of `double` values in C++:

```
// Stack
double values[100];
// Heap
double *values = new double[100];
delete []values;
// C99 style brace enclosed lists
double values[100] = {0}; // Set all to 0
double values[100] = {1, 2, 3}; // 1,2,3,0,0,0,0...
// C99 with designator
double values[100] = {1, 2, 3, [99] 99}; // 1,2,3,0,0,0,...,0,99
```

And in Rust:

```
// Stack
let mut values = [0f64; 100]; // 100 elements
let mut values = [1f64, 2f64, 3f64]; // 3 elements 1,2,3
// Heap
let mut values = Box::new([0f64; 100]);
```

Note how Rust provides a shorthand to initialise the array with the same value or assigns the array with every value. Initialisation in C and C++ is optional however it is more expressive in that portions of the array can be set or not set using enclosed list syntax.

Rust actually *forces* you to initialise an array to something. Attempting to declare an array without assigning it a value is a compiler error.

Slices

A slice is a runtime view of a part of an array or string. A slice is not a copy of the array / string rather that it is a reference to a portion of it. The reference holds a pointer to the starting element and the number of elements in the slice.

```
let array = ["Mary", "Sue", "Bob", "Michael"];
println!("{:?}", array);
let slice = &array[2..];
println!("{:?}", slice);
```

This slice represents the portion of array starting from index 2.

```
["Mary", "Sue", "Bob", "Michael"]
["Bob", "Michael"]
```

Size of the array

C and C++ basically give no easy way to know the length of the array unless you encapsulate the array with a `std::array` or happen to remember it from the code that declares it.

```
// C++11
std::array<Element, 100> elements;
std::cout << "Size of array = " << elements.size() << std::endl;
```

The `std::array` wrapper is of limited use because you cannot pass arrays of an unknown size to a function. Therefore even with this template you may pass the array into a function as one argument and its size as another.

Alternatively you might see code like this:

```
const size_t num_elements = 1024;
char buffer[num_elements];
//...
// fill_buffer needs to be told how many elements there are
fill_buffer(buffer, num_elements);
```

Or like this

```
Element elements[100];
//...
int num_elements = sizeof(elements) / sizeof(Element);
```

In Rust, the array has a function bound to it called `len()`. This always provides the length of the array. In addition if we take a slice of the array, that also has a `len()`.

```
let buffer: [u8; 1024]
println!("Buffer length = {}", buffer.len());

fill_buffer(&buffer[0..10]);
//...
fn fill_buffer(elements: &[Element]) {
  println!("Number of elements = {}", elements.len());
}
```

Strings

Strings in C++ are a bit messy thanks to the way languages and characters have been mapped onto bytes in different ways. The explanation for this requires some backstory...

What is a character exactly?

Historically in C and C++, a char type is 8-bits. Strictly speaking a char is signed type (usually -128 to 127), but the values essentially represent the values 0-255.

The US-ASCII standard uses the first 7-bits (0-127) to assign values to upper and lower case letters in the English alphabet, numbers, punctuation marks and certain control characters.

It didn't help the rest of the world who use different character sets. And even ASCII was competing with another standard called EBDIC which was found on mainframe computers.

What about the upper values from 128-255? Some operating systems came up with a concept called a "code page". According to what "code page" was in effect, the symbol that the user would see for a character in the 128-255 range would change.

But even this is not enough. Some languages like Chinese, Japanese, Korean, Thai, Arabic etc. have thousands of symbols that must be encoded with more than one byte. So the first byte might be a modifier that combines with further bytes to render as a symbol. For example Microsoft's code page 932 use an encoding called Shift JIS (Japanese) where some symbols are two bytes.

Obviously this was rapidly becoming a mess. Each code page interpretted the same byte array differently according to some external setting. So you could not send a file written in Chinese to someone with a different code page and expect it to render properly.

Unicode to the rescue

The Unicode standard assigns every printable symbol in existence with a unique 32-bit value, called a code point. Most symbols fall in the first 16-bits called the Basic Multilingual Plane (BMP).

China has mandated all software must support all 32-bits. We'll see how this has become a major headache for C and C++

C / C++

There is no string primitive

C and C++ does not have a string primitive type, instead it has `char` type, that is one byte. A "string" is a pointer to an array of chars that are terminated with a zero byte, `'\0'` .

```
// The array that my_string points at ends with a hidden \0
char *my_string = "This is as close to a string primitive as you can get";
```

In C, functions such as `strlen()` , `strcpy()` , `strdup()` etc. allow strings to be manipulated but they work by using the zero byte to figure out the length of things. So `strlen()` the number of bytes that were encountered before a `\0` was found. Sicne these functions run until they find a terminating character it is very easy to accidentally for them to overrun a buffer.

In C++ the `std::string` class wraps a char pointer and provides safe methods for modifying the string in a safe manner. It is a vast improvement over C but it is still not a primitive - it is a class defined in a header that is compiled and linked to the executable just like every other class.

In addition, a `std::string` will usually use heap to store the string's data which can have repercussions for memory usage and fragmentation. There is usually a hidden cost to assigning one string to another because memory must be allocated to receive a copy of the string, even if the string itself is not modified during the assignment.

Unicode support

C/C++ added Unicode support by creating a wide character called `wchar_t` . And C++ has an equivalent `std::wstring` .

We're sorted now right?

Oops no, because `wchar_t` type can be either 2 or 4 bytes wide and is a compiler / platform specific decision.

In Microsoft Visual C++ the wide char is an `unsigned short` (corresponding to Win32's Unicode API), in gcc it can be 32-bits or 16-bits according to the compile flags.

A 16-bit value will hold symbols from the Basic Multilingual Plane but not the full 32-bit range. This means that 16-bit wide strings should be assumed to be UTF-16 encoded because they cannot support Unicode properly otherwise.

C++11 rectifies this by introducing explicit `char16_t` and `char32_t` types and corresponding versions of string called `std::u16string` and `std::u32string`.

Character types

So now C++ has 4 character types. Great huh?

Character type	Encoding
`char`	C, ASCII, EBDIC, UTF-8, ad hoc, ???
`wchar_t`	UTF-16 or UTF-32
`char16_t`	UTF-16
`char32_t`	UTF-32

Rust

Rust has been rather fortunate. Unicode preceded it so it makes a very simple design choice.

- A `char` type is a 32-bit Unicode character, always enough to hold a single character.
- A `str` type is a UTF-8 encoded string held in memory. Code tends to use &str which is a string slice, basically a reference to the str, or a portion of it. A str does not need to be terminated with a zero byte and can contain zero bytes if it wants.
- A `std::String` is a heap allocated string type use for manipulating strings, building them, reading them from file, cloning them etc.

Note that internally UTF-8 is used for encoding yet a char is 32-bits. The length of a strings is considered to be its byte length. There are special iterators for walking the string and decoding UTF-8 into 32-bit characters.

Finally there is a platform specific type `OSString` that handles any differences in how the operating system sees strings compared to Rust.

Types Comparison

C++	Rust
	`char *` or `wchar_t *`
C++11 - `char16_t *` , `char32_t *`	`str` , `&str`
	`std::string` , `std::wstring`
`std::u16string` `std::u32string`	`std::String`

char * vs str

C/C++ do not have a string primitive. A string is a pointer to some bytes in memory that are nul terminated. The same applies for wider chars, except of course they require 2 or 4 bytes.

```
// The traditional way
char *my_str = "Hello"; // Terminates with \0
// or
char my_str[] = "Hello"; // Terminates with \0
// or wide string with L prefix
wchar_t hello_chinese = L"\u4f60\u597d";
// C11 and C++11 add UTF string literal prefixes
auto hello_8  = u8"\u4f60\u597d"; // UTF-8 encoded
auto hello_16 =  u"\u4f60\u597d"; // UTF-16
auto hello_32 =  U"\u4f60\u597d"; // UTF-32
```

Rust would use a `str` for this purpose. A `str` is an *immutable* array of bytes somewhere in memory. The `str` could be on the heap when it points to a `String` object, or it could be in global memory if the string is static. A str *slice* is `&str`, is reference to a str which also contains a length value.

```
let my_str = "Hello";
let hello_chinese = "你好";
```

Type inferences for these assignments will create a string slice pointing to the statically allocated string data. The data itself doesn't move and the `&str` is read-only.

We can also observe that Rust removes the mess of character width and literal prefixes that C and C++ have to suffer under because Unicode characters are implicitly supported.

The `str` has functions for iterating over the string in bytes / characters, splitting, find a pattern etc.

```
let my_str = "Hello"; // v is a &'static str
println!("My string is {} and it is {} bytes long", v, v.len());
```

Note `len()` is the length in bytes because strings are UTF-8 encoded. A single character may be encoded as 1, 2, 3, or 4 bytes. It may not be the number of characters a human would actually see.

```
let my_str = "你好";
println!("Number of bytes = {}", my_str.len());
println!("Number of chars = {}", my_str.chars().count());
```

```
Number of bytes = 6
Number of chars = 2
```

You can split a `&str` to produce a left and a right `&str` slice like this:

```
let (part1, part2) = "Hello".split_at(3);
println!("Part 1 = {}", part1);
println!("Part 2 = {}", part2);
```

```
Part 1 = Hel
Part 2 = lo
```

std::basic_string (C++) vs std::String (Rust)

The standard C++ library also has template class `std::basic_string` that acts as a wrapper around the various character types and can be used for manipulating a string of any width. This template is specialised as `std::string`, `std:wstring`, `std::u16string` and `std::u32string`.

```
std::string my_str = "Hello";
my_str += " world";

// C++11 also allows some type inference with autos
auto s1 =   "Hello"s; // std::string
auto s2 = u8"Hello"s; // std::string, forces UTF-8 encoding
auto s3 = L"Hello"s;  // std::wstring
auto s4 = u"Hello"s;  // std::u16string
auto s5 = U"Hello"s;  // std::u32string
```

In Rust, the `std::String` type serves the same purpose:

```
let v = String::from("Hello");
v.push_str(" world");
```

Using it is fairly simple

```
let mut v = String::from("This is a String");
v.push_str(" that we can modify");
```

A `String` has functions to do actions such as appending, e.g.

```
let b = String::from(" Bananas");
let mut result = String::new();
result.push_str("Apples ");
result.push('&'); // Push a char
result.push_str(b.as_str());
println!("result = {}", result);
```

Strings are always valid UTF-8.

Internally a String has a pointer to the data, its length and a capacity (max size). If you intend to expand a string, then you should ensure the `String` has sufficient capacity to accommodate its longest value otherwise you may cause it to reallocate itself excessively.

Strings will never shrink their capacity unless you explicitly call `shrink_to_fit()`. This means if you use a temporary string in a loop, it's probably best to place it outside the loop and reserve space to make it efficient.

```
let mut v = String::with_capacity(100);
// or
let mut v = String::new();
v.reserve_exact(100);
```

Strings also have all the methods of str thanks to implementing `Deref` trait.

Formatting strings

Strings can be formatted in C with `printf` or `sprintf` or in C++ composed with stream operators, e.g. to a `std::stringstream`.

Rust uses `format!` and `println!` macros that more resemble the `sprintf` model.

C++	Rust formatting trait	Purpose
`%s`, `%u`, `%d`, `%i`, `%f`, `%c`	`{}`	C/C++ require the type of the parameter to be specified. In Rust the type is inferred and `{}` will invoked the type's Display trait regardless of what it is. So a String outputs its text, numeric types return their value, boolean as returns true or false, and so on.
`%lld`, `%llu`	`{}`	C/C++ has extensions to deal with different size ints and floats, e.g. ll for long long due to the way arguments are passed to the function. In Rust, there is no need for that.
	`{:?}`, `{:#?}`	In Rust `{:?}` returns whatever is implemented by a type's Debug trait. The `{:#?}` variant can be used to pretty-print the output for types that derive the Debug trait.
`%-10s`	`{:<10}`	Format left aligned string padded to minimum of 10 spaces
`%04`	`{:04}`	Pad a number with zero's to a width of 4
`%.3`	`{:.3}`	Pad a number's precision to 3 decimal places. May also be zero-padded, e.g. {:.03}
`%e`, `%E`	`{:e}`, `{:E}`	Exponent in lower or uppercase
`%x`, `%X`	`{:x}`, `{:X}`	Hexadecimal in lower or uppercase. Note `{:#x}`, `{:#X}` prefixes the output with 0x
`%o`	`{:o}`	Octal. Note `{:#o}` prefixes the output with 0o
	`{:b}`	Binary. Note `{:#b}` prefixes the output with 0b
`%p`	`{:p}`	Presents a struct's memory location, i.e. pointer

Rust has many more formatting traits than this.

For example it allows named parameters like this example:

```
let message = format!("The temperature {temp}C is within
{percent} of maximum", temp = 104, percent = 99);
```

Named parameters would be particularly useful for localization where the order of values may be different in one language compared to another.

Display and Debug traits

Rust allows types to be formatted as strings based upon the formatting traits they implement.

The two main implementation traits are:

- `Display` - this is for standard textual representation of a type.
- `Debug` - this is for the debugging textual representation of a type. It might present additional information or be formatted separately to the Display trait. It is possible to `#[derive(Debug)]` this trait which is usually enough for the purpose of debugging.

If we look at the traits we can see they're identical

```
// std::fmt::Display
pub trait Display {
    fn fmt(&self, &mut Formatter) -> Result<(), Error>;
}
// std::fmt::Debug
pub trait Debug {
    fn fmt(&self, &mut Formatter) -> Result<(), Error>;
}
```

All of the intrinsic types implement `Display` and `Debug`. We can explicitly implement Display on our own structs too:

```
use std::fmt::{self, Formatter, Display};

struct Person {
  first_name: String,
  last_name: String,
}

impl Display for Person {
  fn fmt(&self, f: &mut Formatter) -> fmt::Result {
    write!(f, "{} {}", self.first_name, self.last_name)
  }
}
//...
let person = Person { first_name: "Susan".to_string(),
last_name: "Smith".to_string() };
println!("Person - {}", person);
```

```
Person - Susan Smith
```

Implementing `Debug` is usually done by `#[derive(Debug)]` but it could also be implemented. The derived `Debug` will print out the struct name, and then the members in curly braces:

```
#[derive(Debug)]
struct Person {
  //...
}
//...
println!("Person - {:?}", person);
```

```
Person - Person { first_name: "Susan", last_name: "Smith" }
```

Many types process formatting traits which are values held between the `{}` braces in the string. These are fairly similar to the patterns used in C functions for printf, sprintf etc.

OsString / OsStr

Rust recognises there are times when you need to pass or receive a string from a system API.

In this case you may use `OsString` which allows interchange between Rust and a system dependent representations of strings. On Windows it will return UTF-16 strings, on Linux / Unix systems it will return UTF-8.

An `OsStr` is a slice onto `OsString`, analogous to `str` and `String`.

Variables

C++

Type Inference

C++11 has type inference, previous versions of C++ do not. Type inference allows the programmer to assign a value to an `auto` typed variable and let the compiler infer the type based on the assignment.

Boolean and numeric types are fairly easy to understand providing the code is as explicit as it needs to be.

```
auto x = true; // bool
auto y = 42;   // int
auto z = 100.; // double
```

Where C++ gets messy is for arrays and strings. Recall that strings are not primitive types in the strong sense within C or C++ so auto requires they be explicitly defined or the type will be wrong.

```
auto s = std::string("Now is the window of our discontent"); //
char string
auto s = U"Battle of Waterloo"; // char32_t pointer to UTF-32
string literal
```

Strings are covered elsewhere, but essentially there are many kinds of strings and C++/C has grown a whole bunch of string prefixes to deal with them all.

Arrays are a more interesting problem. The `auto` keyword has no easy way to infer array type so is one hack workaround to assign a templatized array to an `auto` and coerce it.

```
template <typename T, int N> using raw_array = T[N];
auto a = raw_array<int, 5>{};
```

Rust

Rust, variables are declared with a `let` command. The `let` may specify the variable's type, or it may also use type inference to infer it from the value it is assigned with.

```
let x = true; // x: bool
let y = 42; // y: i32
let z = 100.0; // z: f64
let v = vec![10, 20, 30]; // v: Vec<i32>
let s = "Now is the winter of our discontent".to_string(); // s: 
String
let s2 = "Battle of Waterloo"; // s2: &str
let a1: [i32; 5] = [1, 2, 3, 4, 5];
```

Rust has no problem with using type inference in array assignments:

```
let a2 = ["Mary", "Fred", "Sue"];
```

Note that all array elements must be the same type, inference would generate a compiler error for an array like this:

```
// Compile error
let a3 = ["Mary", 32, true];
```

Scope rules

Scope rules in C, C++ and Rust are fairly similar - the scope that you declare the item determines its lifetime.

Shadowing variables

One very useful feature of Rust is that you can declare the same named variable more than once in the same scope or nested scopes and the compiler doesn't mind. In fact you'll use this feature a lot.

This is called *shadowing* and works like this:

```
let result = do_something();
println!("Got result {:?}", result);
if let Some(result) = result {
  println!("We got a result from do_something");
}
else {
  println!("We didn't get a result from do_something");
}

let result = do_something_else();
//...
```

This example uses the variable name `result` 3 times. First to store the result of calling `do_something()`, then to extract some value `Foo` from `Option<Foo>` and a third time for calling something else. We could have assigned `result` to `result2` and then later on assigned the value `do_something_else()` to `result3` but we didn't need to because of shadowing.

Pointers

In C++

A pointer is a variable that points to an address somewhere in memory. The pointer's *type* indicates to the compiler what to expect at the address but there is no enforcement to ensure that the address actually holds that type. A pointer might might be assigned `NULL` (or `nullptr` in C++11) or may even be garbage if nothing was assigned to it.

```
char *name = "David Jones";

int position = -1;
find_last_index("find the letter l", 'l', &position);
```

Generally pointers are used in situations where references cannot be used, e.g. functions returning allocated memory or parent / child collection relationships where circular dependencies would prevent the use of references.

C++11 deprecates `NULL` in favour of new keyword `nullptr` to solve a problem with function overloading.

```
void read(Data *data);
void read(int value);
// Which function are we calling here?
read(NULL);
```

Since `NULL` is essentially `#define NULL 0` and 0 is an integer, we call the wrong function by accident. So C++ introduces an explicit `nullptr` for this purpose.

```
read(nullptr);
```

In Rust:

Rust supports pointers, normally called *raw* pointers however you will rarely use them unless you need to interact with C API or similar purposes.

A pointer looks fairly similar to that of C++:

```
// This is a reference coerced to a const pointer
let age: u16 = 27;
let age_ptr: *const u16 = &age;

// This is a mut reference coerced to a mutable pointer
let mut total: u32 = 0;
let total_ptr: *mut u32 = &mut total;
```

Although you can make a pointer outside of an unsafe block, many of the functions you might want to perform on pointers are unsafe by definition and must be inside `unsafe` blocks.

The documentation in full is here.

References

In C++

A reference is also a variable that points to an address but unlike a pointer, it cannot be reassigned and it cannot be `NULL`. Therefore a reference is generally assumed to be safer than a pointer. It is still possible for the a reference to become dangling, assuming the address it referenced is no longer valid.

In Rust

A reference is also lifetime tracked by the compiler.

Tuples

A tuple is list of values held in parenthesis. They're useful in cases where transient or ad-hoc data is being passed around and you cannot be bothered to write a special struct just for that case.

In C++

C++ does not natively support tuples, but C++11 provides a template for passing them around like so:

```
#include <tuple>

std::tuple<int, int> get_last_mouse_click() {
  return std::make_tuple(100, 20);
}

std::tuple<int, int> xy = get_last_mouse_click();
int x = std::get<0>(xy);
int y = std::get<1>(xy);
```

In Rust

Tuples are part of the language and therefore they're far more terse and easy to work with.

```
fn get_last_mouse_click() -> (i32, i32) {
  (100, 20)
}
// Either
let (x, y) = get_last_mouse_click();
println!("x = {}, y  = {}", x, y);
// or
let xy = get_last_mouse_click();
println!("x = {}, y  = {}", xy.0, xy.1);
```

Literals

C++

Integers

Integer numbers are a decimal value followed by an optional type suffix.

In C++ an integer literal can be expressed as just the number or also with a suffix. Values in hexadecimal, octal and binary are denoted with a prefix:

```
// Integers
42
999U
43424234UL
-3456676L
329478923874927ULL
-80968098606968LL
// C++14
329'478'923'874'927ULL
// Hex, octal, binary
0xfffe8899bcde3728 // or 0X
07583752256
0b111111110010000 // or 0B
```

The `u` , `l` , and `ll` suffixes on integers denotes if it is `unsigned` , `long` or a `long long` type. The `u` and `l` / `ll` can be upper or lowercase. Ordinarily the `u` must precede the size but C++14 allows the reverse order.

C++14 also allows single quotes to be inserted into the number as separators - these quotes can appear anywhere and are ignored.

Floating point numbers

Floating point numbers may represent whole or fractional numbers.

Boolean values

C/C++ `bool` literals are `true` or `false`.

Characters and Strings

A character literal is enclosed by single quotes and an optional width prefix. The prefix `L` indicates a wide character, `u` for UTF-16 and `U` for UTF-32.

```
'a'
L'a' // wchar_t
u'\u20AC' // char16_t
U'\U0001D11E' // char32_t
```

One oddity of a `char` literal is that `sizeof('a')` yields `sizeof(int)` in C but `sizeof(char)` in C++. It isn't a good idea to test the size of a character literal.

A `char16_t` and `char32_t` are sufficient to hold any UTF-16 and UTF-32 code unit respectively.

A string is a sequence of characters enclosed by double quotes. A zero value terminator is always appended to the end. Prefixes work the same as for character literals with an additional `u8` type to indicate a UTF-8 encoded string.

```
"Hello"
u8"Hello" // char with UTF-8
L"Hello"   // wchar_t
u"Hello"   // char16_t with UTF-16
U"Hello"   // char32_t with UTF-32
```

User-defined literals

C++11 introduced user-defined literals. These allow integer, floats, chars and strings to have a user defined type suffix consisting of an underscore and a lowercase string. The prefix may act as a form of decorator or even a constant expression operator which modifies the value at compile time.

C++14 goes further and defines user-defined literals for complex numbers and units of time.

See the link for more information.

Rust

Integers

In Rust number literals can also be expressed as just the number or also with a suffix. Values in hexadecimal, octal and binary are also denoted with a prefix:

```
// Integers
123i32;
123u32;
123_444_474u32;
0usize;
// Hex, octal, binary
0xff_u8;
0o70_i16;
0b111_111_11001_0000_i32;
```

The underscore in Rust is a separator and functions the same way as the single quote in C++14.

Floating point numbers

Floating point numbers may represent whole or fractional numbers. As with integers they may be suffixed to indicate their type.

```
let a = 100.0f64;
let b = 0.134f64;
let c = 2.3f32; // But 2.f32 is not valid (note 1)
let d = 12E+99_E64;
```

One quirk with floating point numbers is the decimal point is used for float assignments but it's also used for member and function invocation. So you can't say `2.f32` since it thinks you are referencing f32 on 2. Instead syntax requires you to say `2.f32` or alter how you declare the type, e.g. `let v: f32 = 2.;`.

Booleans

Boolean literals are simply `true` or `false`.

```
true
false
```

Characters and Strings

A character in Rust is any UTF-32 code point enclosed by single quotes. This value may be escaped or not since .rs files are UTF-8 encoded.

A special prefix `b` may be used to denote a byte string, i.e. a string where each character is a single byte.

```
'x'
'\'' # Escaped single quote
b'&' # byte character is a u8
```

Strings are the string text enclosed by double quotes:

```
"This is a string"
b"This is a byte string"
```

The prefix `b` denotes a byte string, i.e. single byte characters. Rust allows newlines, space, double quotes and backslashes to be escaped using backslash notation similar to C++.

```
"This is a \
  multiline string"
"This string has a newline\nand an escaped \\ backslash"
"This string has a hex char \x52"
```

Strings can also be 'raw' to avoid escaping. In this case, the string is prefixed r followed by zero or more hash marks, the string in double quotes and the same number of hash marks to close. Byte strings are uninterpretted byte values in a string.

```
r##"This is a raw string that can contain \n, \ and other stuff
without escaping"##
br##"A raw byte string with "stuff" like \n \, \u and other
things"##
```

Collections

A collection is something that holds zero or more elements in some fashion that allows you to enumerate those elements, add or remove elements, find them and so forth.

- Vector - a dynamic array. Appending or removing elements from the end is cheap (providing the array is large enough to accomodate an additional item). Inserting items or removing them from any other part of the array is more expensive and involves memory movements. Generally speaking you should always reserve enough space in a vector for the most elements you anticipate it will hold. Reallocating memory can be expensive and lead to fragmentation.
- Vecdeque - a ring buffer array. Items can be added or removed from either end relatively cheaply. Items in the array are not arranged sequentially so there is a little more complexity to managing wraparound and removal than a Vector.
- LinkedList - a linked list individually allocates memory for each element making it cheap to add or remove elements from anywhere in the list. However there is a lot more overhead to iterating the list by index and much more heap allocation.
- Set - a collection that holds a unique set of items. Inserting a duplicate item will not succeed. Some sets maintain the order of insertion. Sets are useful where you want to weed out duplicates from an input.
- Map - a collection where each item is referenced by a unique key. Some maps can maintain the order of insertion.

C++ and Rust have have collections as part of their standard library as is common with modern languages.

C	C++	Rust
-	`std::vector`	`std::vec::Vec` or `std::collections::VecDeque`
-	`std::list`	`std::collections::LinkedList`
-	`std::set`	`std::collections::HashSet` , `std::collections::BTreeSet`
-	`std::map`	`std::collections::HashMap` , `std::collections::BTreeMap`

C has no standard collection classes or types. Some libraries offer collection APIs such as glib or cii.

Iterators

Iterators are a reference to a position in a collection with the means to step through the collection one element at a time.

C++

C++11 provides a shorthand way of iterating a collection:

```
std::vector<char> values;
for (const char &c : values) {
    // do something to process the value in c
}
```

Iterators are more explicit in C++98 and before and the code in C++11 is basically equivalent to this:

```
std::vector<char> values;

for (std::vector<char>::const_iterator i = values.begin(); i !=
values.end(); ++i) {
    const char &c = *i;
    // do something to process the value in c
}
```

This is quite verbose, but essentially each collection type defines a mutable `iterator` and immutable `const_iterator` type and calling `begin` returns an iterator to the beginning of the collection. Calling the `++` operator overload on the iterator causes it to advance to the next element in the collection. When it hits the exclusive value returned by `end` it has reached the end of the collection.

Obviously with an indexed type such as a `vector` you could also reference elements by index, but it is far more efficient to use iterators for other collection types.

Processing collections

C++ provides a number of utility templates in for modifying sequences in collections on the fly.

Rust

Rust also has iterators which work in a similar fashion to C++ - incrementing their way through collections.

TODO

TODO chaining iterators together

TODO mapping one collection to another collection

Structs

C++

A `class` and a `struct` in C++ are largely the same thing from an implementation standpoint. They both hold fields and they both can have methods attached to the class (`static`) or instance level.

```
class Foo {
public:
   // Methods and members here are publicly visible
   double calculateResult();
protected:
   // Elements here are only visible to ourselves and subclasses
   virtual double doOperation(double lhs, double rhs);
private:
   // Elements here are only visible to ourselves
   bool debug_;
};
```

The default access level is `public` for struct and `private` for class. Some rules about templates only apply to classes.

From a psychological perspect a `struct` tends to be used to hold public data that is largely static and/or passed around. A `class` tends to be something more self contained with methods that are called to access or manage private fields.

So these are equivalents:

```
struct Foo { // as a struct
private:
};

class Foo { // As a class
};

// Or the other way around

struct Bar {
};

class Bar {
public:
};
```

Classes can also use an access specifier to inherit from a base class. So a class may specify `public`, `protected` or `private` when deriving from another class depending on whether it wants those methods to be visible to callers, or subclasses.

Classes and structs may have special constructor and destructor methods which are described in sections below.

```
class Size {
public:
  Size(int width, int height);

  int width_;
  int height_;

  int area() const;
};
```

Then in the .cpp file you might implement the constructor and method:

```
Size::Size(int width, int height) : width_(width),
height_(height) {}

int Size::area() { return width_ * height_; }
```

Rust

Rust only has structs. A `struct` consists of a definition which specifies the fields and their access level (public or not), and an `impl` section which contains the implementation of functions bound to the struct.

```
struct Size {
  pub width: i32;
  pub height: i32;
}
```

An `impl` section follows containing the associated functions:

```
impl Size {
  pub fn new(width: i32, height: i32) -> Size {
    Size { width: width, height: height, }
  }

  pub fn area(&self) -> i32 {
    self.width * self.height
  }
}
```

The `new()` function here is a convenience method that returns a struct preinitialised with the arguments supplied. The `area()` function specifies a `&self` argument and returns an area calculation. Any function that supplies a `&self`, or `&mut self` can be called from the variable bound to the struct.

```
let size = Size::new(10, 20);
println!("Size = {}", size.area());
```

The `self` keyword works in much the same way as C++ uses `this`, as a reference to the struct from which the function was invoked. If a function modifies the struct it must say `&mut self`, which indicates the function modifies the struct.

There is no inheritance in Rust. Instead, a `struct` may implement zero or more traits. A trait describes some kind of behavior that can be associated with the struct and described further later on in this chapter.

Constructors

In C++ all classes have implicit or explicit constructors. Either the compiler generates them or you do, or a mix of both.

An implicit default constructor, copy constructor and assignment operator will be created when a class does not define its own. We saw on page 73 why this could be really bad news.

What becomes obvious from reading there is a lot of noise and potential for error in C++. There would be even more if raw pointers were used instead of a `std::unique_ptr` here.

In Rust, things are simpler, and we'll see how it shakes out errors.

First off, let's declare our equivalent struct in Rust:

```
struct Person {
  pub name: String,
  pub age: i32,
  pub credentials: Option<Credentials>,
}
```

Since credentials are optional, we wrap in an `Option` object, i.e. credentials might be None or it might be `Some(Credentials)`. Any code anywhere in the system can instantiate a `Person` simply be declaring an instance:

```
let person = Person { name: String::from("Bob"), age: 20,
credentials: None }
```

In Rust you cannot create a struct without initialising all its members so we cannot have a situation where we don't know what is in each field - it MUST be set by our code.

But declaring the struct is a bit clumsy, especially if the struct is created in lots of places. So can write function that behaves like a constructor in C++.

Instead you implement a static method in the impl of the Struct which returns an initialised struct, e.g.

```
impl Person {
  pub fn new(name: String, age: String) -> Person {
    Person { name: name.clone(), age: age, credentials: None }
  }
}
```

Note that Rust does not support overloads. So if we had multiple "constructor" methods, they would each have to have unique names.

Finally what is we wanted to copy the `Person` struct?

By default Rust does not allow copying on user-defined structs. Assigning a variable to another variable moves ownership, it doesn't copy.

There are two ways to make a user-defined struct copyable

1. implement the `Copy` trait which means assignment is implicit, but is what we want? Do we really want to make copies of a struct by accident?
2. implement `Clone` instead to add a `clone()` method and require an explicit call to `clone()` order to duplicate the struct a copy.

But the compiler can derive clone() providing all the members of the struct implement the Clone trait.

```
#[derive(Clone)]
struct Person {
  pub name: String,
  pub age: i32,
  pub credentials: Option<Credentials>, // Credentials must
implement Clone
}

impl Person {
  pub fn new(name: String, age: String) -> Person {
    Person { name: name.clone(), age: age, credentials: None }
  }
}

//...

let p = Person::new(String::from("Michael"), 20);
let p2 = p.clone();
```

What we can see is that Rust's construction and `clone()` behavior is basically declarative. We saw how C++ has all kinds of rules and nuances to construction, copy construction and assignment which make it complicated and prone to error.

Destructors

A C++ destructor is a specialized method called when your object goes out of scope or is deleted.

```
class MyClass {
public:
  MyClass() : someMember_(new Resource()) {}
  ~MyClass() {
     delete someMember_;
  }

private:
  Resource *someMember_;
}
```

In C++ you can declare a class destructor to be called when the object is about to be destroyed. You have to use a virtual destructor if your class inherits from another class in case a caller calls `delete` on the base class.

Since Rust does not do inheritance and does not have constructors, the manner in which you cleanup is different and simpler. Instead of a destructor you implement the `Drop` trait.

```
impl Drop for Shape {
    fn drop(&mut self) {
        println!("Shape dropping!");
    }
}
```

The compiler recognizes this trait. If you implement this trait then the compiler knows to call your `drop()` function prior to destroying your struct. It's that simple.

Occasionally there might be a reason to explicitly drop a struct before it goes out of scope. Perhaps the resources held by the variable should be freed as soon as possible to release a resource which is in contention. Whatever the reason, the answer is to call `drop` like this:

```
{
  let some_object = SomeObject::new();
  //...
  // Ordinarily some_object might get destroyed later,
  // but this makes it explicitly happen here
  drop(some_object);
  //...
}
```

Access specifier rules

A C++ class can hide or show methods and members to any other class, or to things that inherit from itself using the public, private and protected keywords:

- `public` – can be seen by any code internal or external to the class
- `private` – can only be used by code internal to the class. Not even subclasses can access these members
- `protected` – can be used by code internal to the class and by subclasses.

A class may designate another function or class as a friend which has access to the private and protected members of a class.

Rust makes things somewhat simpler.

If you want a struct to be visible outside your module you mark it `pub`. If you do not mark it `pub` then it is only visible within the module and submodules.

```
pub struct Person { /* ... */ }
```

If you want publicaccess a member of a struct (including modifying it if its mutable), then mark it `pub`.

```
pub struct Person {
  pub age: u16,
}
```

If you want something to be able to call a function on your struct you mark it `pub`.

```
impl Person {
  pub fn is_adult(&self) -> bool {
    self.age >= 18
  }
}
```

Functions

Functions can be bound to a struct within an `impl` block:

```
impl Shape {
  pub fn new(width: u32, height: u32) -> Shape {
    Shape { width, height }
  }

  pub fn area(&self) -> i32 {
    self.width * self.height
  }

  pub fn set(&mut self, width: i32, height: i32) {
    self.width = width;
    self.height = height;
  }
}
```

Functions that start with a `&self` / `&mut self` parameter are bound to instances. Those without are bound to the type. So the `new()` function can be called as `Shape::new()`.

Where `&self` is provided, the function is invoked on the instance. So for example:

```
let shape = Shape::new(100, 100);
let area = shape.area();
```

Where `&mut self` is provided it signifies that the function mutates the struct.

Unlike C++, all access to the struct has to be qualified. In C++ you don't publishing_interval: Double, lifetime_count: UInt32, max_keep_alive_count: UInt32, max_notifications_per_publish: UInt32, priority: Bytehave to say `this->foo()` to call foo() from another member of the class. Rust requires code to say unambiguously `self.foo()`.

Static functions

Static functions are merely functions in the `impl` block that do not have `&self` or `&mut self` as their first parameter, e.g.

```
impl Circle {
   fn pi() -> f64 { std::f64::consts:PI }
}
//...
let pi = Circle::pi();
```

In other words they're not bound to an instance of a type, but to the type itself. For example, `Circle::pi()`.

Traits

C++ allows one class to inherit from another. Generally this is a useful feature although it can get pretty complex if you implement multiple inheritance, particularly the dreaded diamond pattern.

As we've found out, Rust doesn't have classes at all – they're structs with bound functions. So how do you inherit code? The answer is you don't.

Instead your struct may implement traits which are a bit like partial classes.

A trait is declared like so:

```
trait HasCircumference {
  fn circumference(&self) -> f64;
}
```

Here the trait `HasCircumference` has a function called `circumference()` whose signature is defined but must be implemented.

A type can implement the trait by declaring and `impl` of it.

```
impl HasCircumference for Size {
  fn circumference(&self) -> i32 {
    2.0 * std::f64::consts::PI * self.radius
  }
}
```

A trait may supply default function implementations. For example, a `HasDimensions` trait might implement `area()` to spare the implementor the bother of doing it.

```
trait HasDimensions {
  fn width(&self) -> u32;
  fn height(&self) -> u32;

  fn area(&self) -> u32 {
    self.width() * self.height()
  }
}
```

Lifetimes

In C++ an object lives from the moment it is constructed to the moment it is destructed.

That lifetime is implicit if you declare the object on the stack. The object will be created / destroyed as it goes in and out of scope. It is also implicit if your object is a member of another object - the lifetime is within the containing object, and the declaration order of other members in the containing object.

However, if you allocate your object via `new` then it is up to you when to `delete`. If you `delete` too soon, or forget to `delete` then you may destabilize your program. C++ encourages using smart pointers that manage the lifetime of your object, tying it to the implicit lifetime of the smart pointer itself - when the smart pointer is destroyed, it deletes the held pointer. A more sophisticated kind of smart pointer allows multiple instances of the same pointer to exist at once, and reference counting is used so that when the last smart pointer is destroyed, it destroyes the pointer.

Even so, C++ itself will not care if you initialized a class with a reference or pointer to something that no longer lives. If you do this, your program will crash.

Let's write an `Incrementor` class which increments an integer value and returns that value.

```
class Incrementor {
public:
    Incrementor(int &value) : value_(value) {}
    int increment() { return ++value_; }

private:
    int &value_;
};
```

This seems fine, but what if we use it like this?

```
Incrementor makeIncrementor() {
  // This is a bad idea
    int value = 5;
    return Incrementor(value);
}
```

This code passes a reference to an `int` into the class constructor and returns the `Incrementor` from the function itself. But when `increment()` is called the reference is dangling and anything can happen.

Rust lifetimes

Rust *does* care about the lifetime of objects and tracks them to ensure that you cannot reference something that no longer exists. Most of the time this is automatic and self-evident from the error message you get if you try something bad.

The compiler also implements a *borrow checker* which tracks references to objects to ensure that:

1. References are held no longer than the lifetime of the object they refer to.
2. Only a single mutable reference is possible at a time and not concurrently with immutable references. This is to prevent data races.

The compiler will generate compile errors if it finds code in violation of its rules.

So let's write the equivalent of `Incrementor` above but in Rust. The Rust code will hold a reference to a integer `i32` and increment it from a bound function:

```
struct Incrementor {
  value: &mut i32
}

impl Incrementor {
  pub fn increment(&mut self) -> i32 {
    *self.value += 1;
    *self.value
  }
}
```

Seems fine, but the first error we get is:

```
2 |   value: &mut u32
  |          ^ expected lifetime parameter
```

We tried to create a struct that manages a reference, but the compiler doesn't know anything about this reference's lifetime and so it has generated a compile error.

To help the compiler overcome its problem, we will annotate our struct with a lifetime which we will call `'a`. The label is anything you like but typically it'll be a letter.

This lifetime label is a hint on our struct that says the reference we use inside the struct must have a lifetime of at least as much the struct itself - namely that `Incrementor<'a>` and `value: &'a mut i32` share the same lifetime constraint and the compiler will enforce it.

```
struct Incrementor<'a> {
  value: &'a mut i32
}

impl <'a> Incrementor<'a> {
  pub fn increment(&mut self) -> i32 {
    *self.value += 1;
    *self.value
  }
}
```

With the annotation in place, we can now use the code:

```
let mut value = 20;
let mut i = Incrementor { value: &mut value };
println!("value = {}", i.increment());
```

Note that the annotation `'a` could be any label we like - `'increment` would work if we wanted, but obviously its more longwinded.

There is a special lifetime called `'static` that refers to things like static strings and functions which have a lifetime as long as the runtime and may therefore be assumed to always exist.

Lifetime elision

Rust allows reference lifetimes to be elided (a fancy word for omit) in most function signatures.

Basically, it assumes that when passing a reference into a function, that the lifetime of the reference is implicitly longer than the function itself so the need to annotate is not necessary.

```
fn find_person(name: &str) -> Option<Person>
// instead of
fn find_person<'a>(name: &'a str) -> Option<Person>
```

The rules for elision are described in the further reference link.

Further reference

Lifetimes are a large subject and the documentation is here.

Comments

Rust comments are similar to C++ except they may contain Unicode because .rs files are UTF-8 encoded:

```
/*
 This is a comment
*/

// This a comment with Unicode, 你好
```

But in addition anything that uses triple slash `///` annotation can be parsed by a tool called `rustdoc` to produce documentation:

```
/// This is a comment that becomes documentation for do_thing
below
pub fn do_thing() {}
/// Returned by server if the resource could not be found
pub const NOT_FOUND = 404;
```

Runnining `cargo doc` on a project will cause HTML documentation to be produced from annotated comments within the file.

Annotation is written in Markdown format. That means you have a human readable language for writing rich-text documentation and if it's not enough you can resort to HTML via tags.

See here for full documentation

Lifetimes, References and Borrowing

When you assign an object to a variable in Rust, you are said to be binding it. i.e your variable "owns" the object for as long as it is in scope and when it goes out of scope it is destroyed.

```
{
  let v1 = vec![1, 2, 3, 4]; // Vec is created
  ...
} // v1 goes out of scope, Vec is dropped
```

So variables are scoped and the scope is the constraint that affects their lifetime. Outside of the scope, the variable is invalid.

In this example, it is important to remember the `Vec` is on the stack but the pointer it allocates to hold its elements is on the heap. The heap space will also be recovered when the `Vec` is dropped.

If we assign v1 to another variable, then all the object ownership is moved to that other variable:

```
{
  let v1 = vec![1, 2, 3, 4];
  let v2 = v1;
  ...
  println!("v1 = {:?}", v1); // Error!
}
```

This may seem weird but it's worth remembering a serious problem we saw in C++, that of copy constructor errors. It is too easy to duplicate a class and inadvertantly share private date or state across multiple instances.

We don't want objects v1 and v2 to share internal state and in Rust they don't. Rust moves the data from v1 to v2 and marks v1 as invalid. If you attempt to reference v1 any more in your code, it will generate a compile error. This compile error will indicates that ownership was moved to v2.

Likewise, if we pass the value to a function then that also moves ownership:

```
{
  let v1 = vec![1, 2, 3, 4];
  we_own_it(v1);
  println!("v = {:?}", v1);
}

fn we_own_it(v: Vec<i32>) {
  // ...
}
```

When we call we_own_it() we moved ownership of the object to the function and it never came back. Therefore the following call using v1 is invalid. We could call a variation of the function called we_own_and_return_it() which does return ownership:

```
v1 = we_own_and_return_it(v1)
...
fn we_own_and_return_it(v: Vec<i32>) -> Vec<i32> {
  // ...
  v1
}
```

But that's pretty messy and there is a better way described in the next section called borrowing.

These move assignments look weird but it is Rust protecting you from the kinds of copy constructor error that is all too common in C++. If you assign a non-Copyable object from one variable to another you move ownership and the old variable is invalid.

If you truly want to copy the object from one variable to another so that both hold independent objects you must make your object implement the Copy trait. Normally it's better to implement the Clone trait which works in a similar way but through an explicit clone() operation.

Variables must be bound to something

Another point. Variables must be bound to something. You cannot use a variable if it hasn't been initialized with a value of some kind:

```
let x: i32;
println!("The value of x is {}", x);
```

It is quite valid in C++ to declare variable and do nothing with it. Or conditionally do something to the variable which confuses the compiler so it only generates a warning.

```
int result;
{
   // The scope is to control the lifetime of a lock
   lock_guard<mutex> guard(data_mutex);
   result = do_something();
}
if (result == 0) {
  debug("result succeeded");
}
```

The Rust compiler will throw an error, not a warning, if variables are uninitialised. It will also warn you if you declare a variable and end up not using it.

References and Borrowing

We've seen that ownership of an object is tracked by the compiler. If you assign one variable to another, ownership of the object is said to have moved to the assignee. The original variable is invalid and the compiler will generate errors if it is used.

Unfortunately this extends to passing values into functions and this is a nuisance. But variable bindings can be borrowed. If you wish to loan a variable to a function for its duration, you can pass a reference to the object:

```
{
  let mut v = Vec::new(); // empty vector
  fill_vector(&mut v);
  // ...
  println!("Vector contains {:?}", v);
}
//...
fn fill_vector(v: &mut Vec<i32>) {
  v.push(1);
  v.push(2);
  v.push(3);
}
```

Here we create an empty vector and pass a mutable reference to it to a function called fill_vector(). The compiler knows that the function is borrowing v and then ownership is returned to v after the function returns.

Expressions

An expression is something that evaluates to something. Just like C++ more or less...

```
let x = 5 + 5; // expression evaluates to 10
```

But blocks are expressions too

Where it gets more interesting is that a block of code, denoted by curly braces also evaluates to an expression. This is legal code:

```
let x = {};
println!("x = {:?}", x);
```

What was assigned to x? In this case the block was empty so x was assigned with the value of `()`. The value `()` is a special unitary type that essentially means neither yes or no. It just means "value". That is the default type of any function or type. It works a little like `void` in C++ meaning the value is meaningless so don't even look at it.

```
x = ()
```

This block also returns a value of `()`.

```
let x = { println!("Hello"); };
println!("x = {:?}", x);
```

Again, that's because although the block does stuff (print Hello), it doesn't evaluate to anything so the compiler returns `()` for us.

So far so useless. But we can change what the block expression evaluates to:

```
let x = {
    let pi = 3.141592735;
    let r = 5.0;
    2.0 * pi * r
};
println!("x = {}", x);
```

Now x assigned with the result of the last line which is an expression. Note how the line is not terminated with a semicolon. That becomes the result of the block expression. If we'd put a semicolon on the end of that line as we did with the println!("Hello"), the expression would evaluate to ().

Use in functions

Trivial functions can just omit the return statement:

```
pub fn add_values(x: i32, y: i32) -> i32 {
  x + y
}
```

You can use return in blocks too

Sometimes you might explicitly need to use the return statement. The block expression evaluates at the end of the block so if you need to bail early you could just use return.

```
pub fn find(value: &str) -> i32 {
  if value.len() == 0 {
    return -1;
  }
  database.do_find(value)
}
```

Simplifying switch statements

In C or C++ you'll often see code like this:

```
std::string result;
switch (server_state) {
  case WAITING:
    result = "Waiting";
    break;
  case RUNNING:
    result = "Running";
    break;
  case STOPPED:
    result = "Stopped";
    break;
  }
}
```

The code wants to test a value in server_state and assign a string to result. Aside from looking a bit clunky it introduces the possibility of error since we might forget to assign, or add a break, or omit one of the values.

In Rust we can assign directly into result of from a match because each match condition is a block expression.

```
let result = match server_state {
    ServerState::WAITING => "Waiting",
    ServerState::RUNNING => "Running",
    ServerState::STOPPED => "Stopped",
};
```

Not only is this half the length it reduces the scope for error. The compiler will assign the block expression's value to the variable result. It will also check that each match block returns the same kind of type (so you can't return a float from one match and strings from others). It will also generate an error if the ServerState enum had other values that our match didn't handle.

Ternary operator

The ternary operator in C/C++ is an abbreviated way to perform an if/else expression condition, usually to assign the result to a variable.

```
bool x = (y / 2) == 4 ? true : false;
```

Rust has no such equivalent to a ternary operator but it can be accomplished using block expressions.

```
let x = if y / 2 == 4 { true } else { false };
```

Unlike C/C++ you could add additiona else ifs, matches or anything else to that providing each branch returns the same type.

Conditions

Conditional code is similar between C++ and Rust. You test the boolean truth of an expression and you can use boolean operators such as && and || to join expressions together.

```
int x = 0;
while (x < 10) {
  x++;
}
int y = 10;
bool doCompare = true;
if (doCompare && x == y) {
  printf("They match!\n");
}
```

In Rust:

```
let mut x = 0;
while x < 10 {
  x = x + 1;
}
let y = 10;
let do_compare = true;
if do_compare && x == y {
  println!("They match!");
}
```

The most notable difference is that Rust omits the outer braces so the code is slightly cleaner. You don't have to omit the outer braces but the compiler will issue a warning if you leave them in.

Ternary operator

The ternary operator is that special ? : shorthand notation you can use to in C++ for simple conditionals.

```
int x = (y > 200) ? 10 : 0;
```

Rust does not support this notation, however you may take advantage of how a block evaluates as an expression to say this instead:

```
let x = if y > 200 { 10 } else { 0 };
```

So basically you can do one line conditional assignments using if and else. Also note that you could even throw in an "else if" or two if that's what you wanted to do:

```
let c = get_temperature();
let water_is = if (c >= 100) { "gas" } else if (c < 0) { "solid"
} else { "liquid" };
```

Conditional "if let"

One unusual feature is the "if let" pattern. This combines a test to see if something matches a pattern and if it does, to automatically assign the result to the tuple. It would be most commonly see in code that returns an enum such as a `Result` or `Option`.

For example:

```
fn search(name: &str) -> Option<Person> { /* ... */ }
//...
if let Some(person) = search("fred") {
  println!("You fould a person {}", person);
}
else {
  println!("Could not find person");
}
```

Switch / Match

C++

A `switch` statement in C or C++ allows a condition or a variable to be compared to a series of values and for code associated with those values to executed as a result. There is also a default clause to match any value that is is not caught explicitly.

```
int result = http_get();
switch (result) {
case 200:
  success = true;
  break;
case 404:
  log_error(result);
  // Drop through
default:
  success = false;
  break;
}
```

Switch statements can be a source of error because behaviour is undefined when a `default` clause is not supplied. It is also possible to inadvertently forget the `break` statement. In the above example, the code explicitly "drops" from the 404 handler into the default handler. This code would work fine providing someone didn't insert some extra clauses between 404 and default...

Additionally switch statements only work on numeric values (or `bool`).

Rust

Match is like a `switch` statement on steroids.

In C++ a switch is a straight comparison of an integer value of some kind (including chars and enums), against a list of values. If the comparison matches, the code next to it executes until the bottom of the switch statement or a break is reached.

TODO

Casting

Casting is the act of coercing one type to be another, or dynamically producing the equivalent value in the other type.

C++ has a range of cast operators that turn a pointer or value of one kind into a pointer or value of another kind.

- `const_cast<T>(value)` - removes the const enforcement from a value so it may be modified.
- `static_cast<T>(value)` - attempts to convert between types using implicit and user defined conversions.
- `reinterpret_cast<T>(value)` - a compiler directive to just treat the input as some other kind. It does not involve any form of conversion.
- `dynamic_cast<T>(value)` - attempts to convert a class pointer / reference to/from other classes in its inheritance hierarchy. Involves runtime checks.
- Traditional C-style cast - a C++ compiler will attempt to interpret it as a `const_cast`, a `static_cast` and a `reinterpret_cast` in varying combinations.

That's a very brief summary of casting which probably invokes more questions than it answers. Casting in C++ is very complex and nuanced. Some casts merely instruct the compiler to ignore const or treat one type as another. A static cast might involve code generation to convert a type. A dynamic cast might add runtime checks and throw exceptions.

Rust has nothing equivalent to this complexity. A numeric type may be converted to another numeric type using the `as` keyword.

```
let a = 123i32;
let b = a as usize;
```

Anything beyond this requires implementing the `Into<>` or `From<>` traits and making conversion an explicit action.

The compiler also does not allow code to cast away `const`-ness or treat one type as another except through `unsafe` code blocks.

Transmutation

Rust allows some types to be transmuted to others. Transmute is an `unsafe` action but it allows a memory location to be treated as another type, e.g. an array of bytes as an integer.

Enumerations

In C++ an `enum` is a bunch of labels assigned an `int` value. i.e. it is basically a bunch of constants with scalar values.

```
enum HttpResponse {
  okay = 200,
  not_found = 404,
  internal_error = 500,
};
```

C++11 extends this concept a little, allowing you to declare an `enum` that uses another kind of integral type, e.g. a `char` to hold the values.

```
enum LibraryCode : char {
  checked_in = 'I',
  checked_out = 'O',
  checked_out_late = 'L'
};
```

In Rust an `enum` can be a scalar value just like in C++.

```
enum HttpResponse {
  Ok= 200,
  NotFound= 404,
  InternalError = 500
};
```

But an enum can also hold actual data so you can convey far more information than a static value could by itself.

```
enum HttpResponse {
  Ok,
  NotFound(String),
  InternalError(String, String, Vec<u8>)
}
```

You can also bind functions to the enum:

```
impl HttpResponse {
  pub fn code(&self) => {
    match *self {
      HttpResponse::Ok => 200,
      HttpResponse::NotFound(_) => 404,
      HttpResponse::InternalError(_, _, _) => 500,
    }
  }
}
```

So we might have a function that makes an http request and returns a response:

```
fn do_request(url: &str) -> HttpResponse {
  if url == "/invalid" {
    HttpResponse::NotFound(url.to_string())
  }
  else {
    HttpResponse::Ok
  }
}
//...
let result = do_request("/invalid");
if let HttpResponse::NotFound(url) = result {
  println!("The url {} could not be found", url);
}
```

Now our code is able to return a more meaningful response in an enum and the code is able to extract that response to print out useful information.

Loops

C++

For loops

A `for` loop in C/C++ consists of 3 expression sections housed in the `for()` section and a block of code to execute:

The three segments of a for statement allow:

- Zero or more variables to be initialized (can be empty)
- Zero or more conditions to be true for the loop to continue (can be empty)
- Zero or more actions to perform on each iteration (can be empty).

So this is a valid for loop:

```
// Infinite
for (;;) {
  //...
}
```

So is this:

```
for (int i = 10, j = 0; (j = i * i) <= 100; i--) {
  //...
}
```

This is clearly a convoluted and somewhat confusing loop because it mixes assignment and conditional tests into the terminating text, but it is one which is entirely legal.

Iterating a range

A C++ loop consists of an initialising expression, a condition expression and a a loop expression separated by semicolons. So a loop that iterates from 0 to 100 looks like this:

```
for (int i = 0; i < 100; i++ ) {
  cout << "Number " << i << endl;
}
```

Iterating C++ collections

C++ introduces the concept of iterators to its collection classes. An `iterator` is something that can increment or decrement to traverse a collection.

So to iterate a collection from one end to the other, an iterator is assigned with the collection's `begin()` iterator and incremented until it matches the `end()` iterator.

```
for (std::vector<string>::const_iterator i = my_list.begin(); i
!= my_list.end(); ++i ) {
  cout << "Value = " << *i << end;
}
```

C++11 provides new range based for-loop with simpler syntax when iterating over arrays and collections:

```
std::vector values;
...
for (const auto & v: values) {
  ...
}

int x[5] = { 1, 2, 3, 4, 5 };
for (int y : x) {
  ...
}
```

Infinite Loop

An infinite loop is one that never ends. The typical way to do this in C++ is to test against an expression that always evaluates to true or use an empty for loop:

```
while (true) {
  poll();
  do_work();
}
// Or with a for loop
for (;;) {
  poll();
  do_work();
}
```

While Loop

C++ has conditional `while() {}` and `do { } while()` forms. The former tests the expression before it even runs while the latter runs at least once before testing the expression.

```
while (!end) {
  std::string next = getLine();
  end = next == "END";
}
```

The do-while form in C++ will execute the loop body at least once because the condition is only tested after each iteration instead of before.

```
int i = 0;
do {
  i = rand();
} while (i < 20);
```

Break and Continue

If you need to exit a loop or start the next iteration early then you use the `break` and `continue` keywords. The break keyword terminates the loop, the continue, causes the loop to proceed to the next iteration.

```
bool foundAdministrator = false;
for (int i = 0; i < loginCredentials; ++i) {
   const LoginCredentials credentials = fetchLoginAt(i);
   if (credentials.disabled) {
     // This user login is disabled so skip it
     continue;
   }
   if (credentials .isAdmin) {
     // This user is an administrator so no need to search rest
of list
     foundAdministrator = true;
     break;
   }
   // ...
}
```

Rust

For loop

Rust's `for` loop is actually sugar over the top of iterators. If a structured type implements the trait `IntoIterator` it can be looped over using a `for` loop.

Basically in pseudo code, the loop desugars to this:

```
If structure type can be turned `IntoIterator`
  Loop
   If let Some(item) = iterator.next() {
     do_action_to_item(item)
   Else
     break;
  End
Else
  Compile Error
Done
```

Iterating a range

A `Range` object in Rust is expressed as `from..to` where `from` and `to` are values or expressions that evaluate to values.

For example:

```
let range=0..33;
// Variables
let min = 0;
let max = 100;
let range2 = min..max;
```

A range is inclusive / exclusive, i.e. the minimum value is included in the `Range` but the maximum value is exclusive.

Here is a simple loop that counts from 0 to 9

```
for i in 0..10 {
  println!("Number {}", i);
}
```

The value `0..10` is a `Range` that runs from 0 to exclusive of 10. A range implements the `Iterator` trait so the for loop advances one element at a time until it reaches the end.

Iterators have a lot of functions on them for doing fancy stuff, but one which is useful in loops is the `enumerate()` function. This transforms the iterator into returning a tuple containing the index and the value instead of just the value.

So for example:

```
for (i, x) in (30..50).enumerate() {
   println!("Index {} is value {}", i, x);
}
```

For loop - Iterating arrays and collections

Here is a loop that iterates an array:

```
let values = [2, 4, 6, 7, 8, 11, 33, 111];
for v in &values {
   println!("v = {}", v);
}
```

Note you can only iterate over an array by reference because iterating it by value would be destructive.

We can directly use the `iter()` function that arrays and collections implement which works by reference:

```
let values = vec![2, 4, 6, 7, 8, 11, 33, 111];
for v in values.iter() {
   println!("v = {}", v);
}
```

If the collection is a map, then iterators will return a key and value tuple

```
use std::collections::HashMap;

let mut values = HashMap::new();
values.insert("hello", "world");
//...
for (k, v) in &values {
  println!("key = {}, value = {}", k, v);
}
```

Another way to iterate is using the `for_each()` function on the iterator itself:

```
let values = [2, 4, 6, 7, 8, 11, 33, 111];
values.iter().for_each(|v| println!("v = {}", v));
```

Break and Continue

Rust also has `break` and `continue` keywords and they operate in a similar fashion - they operate on the innermost loop. A `continue` will start on the next iteration while a `break` will terminate the loop.

```
let values = vec![2, 4, 6, 7, 8, 11, 33, 111];
for v in &values {
  if *v % 2 == 0 {
    continue;
  }
  if *v > 20 {
    break;
  }
  println!("v = {}", v);
}
```

Labels

The `break` and `continue` work by default on the current loop. There will be occasions where you intend to break out of an enclosing loop instead. For those occasions you can label your loops and pass that label into the `break` or `continue:

```
'x: for x in 0..10 {
  'y: for y in 0..10 {
     if x == 5 && y == 5 {
       break 'x;
     }
     println!("x = {}, y = {}", x, y);
  }
}
```

Infinite Loop

Rust has an explicit infinite `loop` that runs indefinitely:

```
loop {
  poll();
  do_work();
}
```

Rust recommends using this form when an infinite loop is required to assist with code generation. Note that an infinite loop can still be broken out of using a `break` statement.

While Loop

A `while` loop in Rust looks pretty similar to one written in C/C++. The main difference is that parentheses are not necessary around the conditional test.

```
while request_count < 1024 {
  process_request();
  request_count = request_count + 1;
}
```

Rust has no equivalent to the do-while loop form. It can be simulated but it looks a bit inelegant:

```
let mut i = 0;
loop {
  i = i + 1;
  if i >= 20 { break; }
}
```

While let loop

Just as there is an `if let` which tests and assigns a value that matches a pattern, there is also a `while let` equivalent:

```
let mut iterator = vec.into_iter();
while let Some(value) = iterator.next() {
  process(value);
}
```

This loop will break when the iterator returns `None` .

Functions

In C++ the standard form of a function is this:

```
// Declaration
int foo(bool parameter1, const std::string &parameter2);

// Implementation
int foo(bool parameter1, const std::string &parameter2) {
  return 1;
}
```

Usually you would declare the function, either as a forward reference in a source file, or in a header. Then you would implement the function in a source file.

If a function does not return something, the return type is `void`. If the function does return something, then there should be return statements for each exiting branch within the function.

You can forego the function declaration in two situations:

1. If the function is inline, i.e. prefixed with the `inline` keyword. In which case the function in its entireity is declared and implemented in one place.
2. If the function is not inline but is declared before the code that calls it in the same source file. So if function `foo` above was only used by one source file, then just putting the implementation into the source would also act as the declaration

In Rust the equivalent to `foo` above is this:

```
fn foo(parameter1: bool, parameter2: &str) -> i32 {
  // implementation
  1
}
```

The implementation *is* the declaration there is no separation between the two. Functions that return nothing omit the `->` return section. The function can also be declared before or after whatever calls it. By default the function is private to the model (and submodules) that implement it but making it `pub fn` exposes it to other modules.

Like C++, the function must evaluate to something for each exiting branch but this is mandatory.

Also note, that the `return` keyword is not usually unecessary. Here is a function that adds two values together and returns them with no return:

```
fn add(x: i32, y: i32) -> i32 {
  x + y
}
```

Why is there no `return`? As we saw in the section on Expressions, a block evaluates to a value if we omit the semi-colon from the end so `x + y` is the result of evaluating the function block and becomes what we return.

There are occasions were you explicitly need the return keyword. Typically you do that if you want to exit the function before you get to the end of the function block:

```
fn process_data(number_of_times: ui32) -> ui32 {
  if number_of_times == 0 {
    return 0;
  }
  let mut result : ui32 = 0;
  for i in number_of_times {
    result += i;
  }
  result
}
```

Variable arguments

C++ functions can take a variable number of arguments with the ... ellipsis pattern. This is used in functions such as print, scanf etc.

```
void printf_like(const char *pattern, ...);
```

Rust does not support variadic functions (the fancy name for this ability). However you could pass additional arguments in an array slice if the values are the same, or as a dictionary or a number of other ways.

TODO Rust example of array slice

Another option is to write your code as a macro. Macros can take any number of expressions so you are able to write code that takes variable arguments. This is how macros such println!, format! and vec! work.

Default arguments

C++ arguments can have default values.

```
std::vector<Record> fetch_database_records(int number_to_fetch =
100);
```

A function defines what its name is, what types it takes and what value (if any) it returns.

Function overloading

C++ functions can be overloaded, e.g.

```
std::string to_string(int x);
std::string to_string(float x);
std::string to_string(bool x);
```

Rust does not support overloading. As an alternative, each variation of the function would have to be named uniquely.

C++11 alternative syntax

C++11 introduces a new syntax which is slightly closer to Rust's in style.

```
auto function_name(type parameter1, type parameter2, ...) ->
return-type;
```

This form was created to allow C++ function declarations to more closely to resemble lambda functions in some scenarios and to help with decltype return values.

Polymorphism

C++

C++ has 4 types of polymorphism:

1. Function name overloading - multiple definitions of the same function taking different arguments.
2. Coercion - implicit type conversion, e.g. assigning a double to an int or a bool.
3. Parametric - compile type substitution of parameters in templates
4. Inclusion - subtyping a class with virtual methods overloads their functionality. Your code can use the pointer to a base class, yet when you call the method you are calling the function implemented by the subtype.

That is to say, the same named function can be overloaded with different parameters.

Function name overloading

```
class Variant {
public:
  void set(); // Null variant
  void set(bool value);
  void set(int value);
  void set(float value);
  void set(Array *value);
};
```

One of the biggest issues that you might begin to see from the above example is that is too easy to inadvertantly call the wrong function because C++ will also implicitly convert types. On top of that C++ also has default parameter values *and* default constructors. So you might call a function using one signature and be calling something entirely different after the compiler resolves it.

```
// Sample code
Variant v;
//...
v.set(NULL);
```

This example will call the integer overload because `NULL` evaluates to 0. One of the changes to `C++11` was to introduce an explicit `nullptr` value and type to avoid this issues.

Rust

Rust has limited support for polymorphism.

1. Function name overloading - there is none. See section below for alternatives.
2. Coercion. Rust allows limited, explict coercion between numeric types using the `as` keyword. Otherwise see below for use on `Into` and `From` traits.
3. Parameteric - similar to C++ via generics
4. Inclusion - there is no inheritance in Rust. The nearest thing to a virtual method in rust is a trait with an implemented function that an implementation overrides with its own. However this override is at compile time.

Alternatives to function name overloading

If you have a few functions you can just disambiguate them, e.g.

```
fn new(name: &str) -> Foo { /* ... */ }
fn new_age(name: &str, age: u16) -> Foo { /* ... */ }
```

Use traits

A common way to do polymorphism is with *traits*.

There are two standard traits for this purpose:

- The `From<>` trait converts from some type into the our type.
- The `Into<>` trait converts some type (consuming it in the process) into our type

You only need to implement `From` or `Into` because one implies the other.

The `From` trait is easier to implement:

```
use std::convert::From;

impl From<&'static str> for Foo {
  fn from(v: &'static str) -> Self {
    Foo { /* ... */ }
  }
}

impl From<(&'static str, u16)> for Foo {
  fn from(v: (&'static str, u16)) -> Self {
    Foo { /* ... */ }
  }
}
//...

let f = Foo::from("Bob");
let f = Foo::from(("Mary", 16));
```

But let's say we want an explicit `new` constructor function on type `Foo`. In that case, we could write it using the `Into` trait:

```
impl Foo {
  pub fn new<T>(v: T) -> Foo where T: Into<Foo> {
    let result = Foo::foo(v);
    // we could code here that we do here after making Foo by
whatever means
    result
  }
}
```

Since `From` implies `Into` we can just call the constructor like so:

```
let f = Foo::new("Bob");
let f = Foo::new(("Mary", 16));
```

If you prefer you could implement `Into` but it's more tricky since it consumes the input, which might not be what you want.

```
// This Into works on a string slice
impl Into<Foo> for &'static str {
    fn into(self) -> Foo {
        //... constructor
    }
}

// This Into works on a tuple consisting of a string slice and a
u16
impl Into<Foo> for (&'static str, u16) {
    fn into(self) -> Foo {
        //... constructor
    }
}

//...
let f: Foo = "Bob".into();
let f: Foo = ("Mary", 16).into();
// OR
let f = Foo::new("Bob");
let f = Foo::new(("Mary", 16));
```

Use enums

Remember that an enumeration in Rust can contain actual data, so we could also implement a function that takes an enumeration as an argument that has values for each kind of value it accepts:

```
pub enum FooCtorArgs {
   String(String),
   StringU16(String, u16)
}

impl Foo {
  pub fn new(v: FooCtorArgs) {
    match v {
      FooCtorArgs::String(s) => { /* ... */ }
      FooCtorArgs::StringU16(s, i) => { /* ... */ }
    }
  }
}
//...
let f = Foo::new(FooCtorArgs::String("Bob".to_string()));
let f = Foo::new(FooCtorArgs::StringU16("Mary".to_string(),
16));
```

Error Handling

C++

C++ allows code to throw and catch exceptions. As the name suggests, exceptions indicate an exceptional error. An exception is thrown to interrupt the current flow of logic and allows something further up the stack which to catch the exception and recover the situation. If nothing catches the throw then the thread itself will exit.

```
void do_something() {
  if (!read_file()) {
    throw std::runtime_error("read_file didn't work!");
  }
}
...
try {
  do_something();
}
catch (std::exception e) {
   std::cout << "Caught exception -- " << e.what() << std::endl;
}
```

Most coding guidelines would say to use exceptions sparingly for truly exceptional situations, and use return codes and other forms of error propagation for ordinary failures.

However C++ has no simple way to confer error information for ordinary failures. Here are some common ways they may work:

- Functions that return a `bool`, an `int`, or a pointer with special meaning. e.g. `false`, `-1` or `NULL` for failure.
- Functions that return a result code or enum. This might have a `GOOD` value and a bunch of `ERROR_` values. An extreme example would be `HRESULT` used by Windows that bitpacks information about goodness, severity and origin into a result and requires macros to extract the information.
- Functions that have a special out parameter that is filled in with additional detail in the case of failure.
- Functions that provide further information about the last error in `errno()` or some similar function supplied by the library.

- Exceptions that are thrown for any failure and must be caught.
- Exceptions that are thrown sometimes and error codes are returned other times.
- Functions that are overloaded into two forms, one that throws an exception, another that stores the error in an error parameter. The boost library has functions like this.

Since there is no consistent way to deal with errors, every library and function has its own ad hoc way to return information.

Rust

Rust provides two enumeration types called `Result` and `Option` that allow functions to propagate results to their caller. The intention is that there are no magic numbers that indicate an error - you either get something or you explicitly get an error / nothing.

It also provides a `panic!()` macro that you can use for unexpected state and other failings in your code. A panic is similar to an exception except there are limits on if you can catch it.

So the normal order of things is:

1. Functions provide a return type which indicates to the caller the success / failure.
2. The caller can propagate the result up the call chain to its caller and so on.
3. For extreme unrecoverable errors there is a panic option.

Result

The type `Result<T, E>` takes a success value type `T` and an error type `E`.

```
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

So perhaps we have `validate_files()` function that either succeeds or it returns with an error. We can define it like so:

```
enum ErrorResultCode {
  ResourcesNotFound(Vec<String>),
  DataCorrupted,
  PermissionDenied
}

fn validate_files() -> Result<(), ErrorResultCode> { /* ... */ }
//...
match validate_files() {
  Ok(_) => { println!("Success"); }
  Err(err) => {
    match err {
      ErrorResultCode::ResourcesNotFound(resources) => {
        println!("Fail resources not found");
        resources.for_each(|resource| println!("Not found {}",
resource));
      }
      ErrorResultCode::DataCorrupted => { println!("Fail data
corrupted"); }
      ErrorResultCode::PermissionDenied => { println!("Fail
permission denied"); }
    }
  }
}
```

The return code `Result<(), ErrorResultCode>` means calling the function will either return:

- `Ok(T)` where the payload is the `()` unity type/value. i.e. when we succeed we get back nothing more of interest.
- `Err(E)` where the payload is `ErrorResultCode` which we can inspect further if we want to.

Option

The `Option` enum either returns `None` or `Some(T)` where the `Some` contains a type `T` payload of data.

This type is particularly useful for functions that either return something or nothing, e.g. a database query.

```
enum Option<T> {
    None
    Some(T)
}
```

For example, we might have a function that searches a database for a person's details, and it either finds them or it doesn't.

```
struct Person { /* ... */}

fn find_person(name: &str) {
    let records = run_query(format!("select * from persons where
name = {}", sanitize_name(name)));
    if records.is_empty() {
        None
    }
    else {
        let person = Person::new(records[0]);
        Some(person)
    }
}
```

The ? directive

Let's say you have 2 functions `delete_user` and `find_user`. The function `delete_user` first calls `find_user` to see if the user even exists and then proceeds to delete the user or return the error code that it got from `find_user`.

```
fn delete_user(name: &str) -> Result<(), ErrorCode> {
  let result = find_user(name);
  if let Ok(user) = result {
     // ... delete the user
     Ok(())
  }
  else {
    Err(result.unwrap_err())
  }
}

fn find_user(name: &str) -> Result<User, ErrorCode> {
  //... find the user OR
  Err(ErrorCode::UserDoesNotExist)
}
```

We have a lot of code in `delete_user` to handle success or failure in `find_user` and throw its failure code upwards. So Rust provides a convenience `?` mark on the end of the call to a function that instructs the compiler to generate the if/else branch we hand wrote above, reducing the function to this:

```
fn delete_user(name: &str) -> Result<(), ErrorCode> {
  let user = find_user(name)?;
  // ... delete the user
  Ok(())
}
```

Providing you want to propogate errors up the call stack, this can eliminate a lot of messy conditional testing in the code and make it more robust.

Older versions of Rust used a special `try!()` macro for this same purpose (not to be confused with `try-catch` in C++) which does the same thing. So if you see code like this, it would be the same as above.

```
fn delete_user(name: &str) -> Result<(), ErrorCode> {
  let user = try!(find_user(name));
  // ... delete the user
  Ok(())
}
```

Nuclear option - panic!()

If code really wants to do something equivalent to a throw / catch in C++ it may call panic!().

This is NOT recommended for dealing with regular errors, only irregular ones that the code has little or no way of dealing with.

This macro will cause the thread to abort and if the thread is the main programme thread, the entire process will exit.

A panic!() can be caught *in some situations* and should be if Rust is being invoked from another language. The way to catch an unwinding panic is a closure at the topmost point in the code where it can be handled.

```
use std::panic;

let result = panic::catch_unwind(|| {
    panic!("Bad things");
});
```

Lambda Expressions / Closures

Lambdas in C++11

A lambda expression, or lambda is an anonymous function that can be declared and passed around from within the scope of the call itself.

This can be particularly useful when you want to sort, filter, search or otherwise do some trivial small action without the bother of declaring and maintaining a separate function.

In C++ a lambda looks like this:

```
float values[10] = { 9, 3, 2.1, 3, 4, -10, 2, 4, 6, 7 };
std::sort(values, values + 10, [](float a, float b) {
  return a < b;
});
```

This lambda is passed to a std::sort function to sort an array of values by some criteria.

A C++ lambda can (but doesn't have to) capture variables from the enclosing scope if it wishes and it can specify capture clauses in the `[]` section that define how capture is made. Captures can be by value or reference, and can explicitly list the variables to capture, or specify to capture everything by reference or assignment. A lambda that captures variables effectively becomes a closure.

```
auto v1 = 10.;
auto v2 = 2.;
// Capture by value
auto multiply = [v1, v2]() { return v1 * v2; };
// Capture by reference
auto sum = [&v1, &v2]() { return v1 + v2; };
cout << multiply() << endl;
cout << sum() << endl;
v1 = 99; // Now v1 in sum() references 99
cout << multiply() << endl;
cout << sum() << endl;
```

We can see from the output that `multiply()` has captured copies of the values in `v1` and `v2`, whereas `sum()` captures by reference and so it is sensitive to changes in the variables:

```
20
12
20
101
```

A capture can also specify a default capture mode by specifying `=` in the capture clause or by reference `&` and then specify capture behaviour for specific variables.

So our captures above could be simplified to:

```
// Capture by value
auto multiply = [=]() { return v1 * v2; };
// Capture by reference
auto sum = [&]() { return v1 + v2; };
```

Note that C++ lambdas can exhibit dangerous behaviour - if a lambda captures references to variables that go out of scope, the lambda's behaviour is undefined. In practice that could mean the application crashes.

Closures in Rust

Rust implements closures. A closure is like a lambda except it automatically captures anything it references from the enclosing environment. i.e. by default it can access any variable that is in the enclosing scope.

Here is the same sort snippet we saw in C++ expressed as Rust. This closure doesn't borrow anything from its enclosing scope but it does take a pair of arguments to compare two values for sorting. The `sort_by()` function repeatedly invokes the closure to sort the array.

```
use std::cmp::Ord;
let mut values = [ 9.0, 3.0, 2.1, 3.0, 4.0, -10.0, 2.0, 4.0,
6.0, 7.0 ];
values.sort_by(|a, b| a < b );
println!("values = {:?}", values);
```

A closure that uses a variable from the enclosing scope borrows it by default. That means the borrowed variable can't change while the closure is in scope. To change the value we must ensure the closure goes out of scope to free the borrow, e.g. with a block:

```
let mut x = 100;
{
  let square = || x * x;
  println!("square = {}", square());
}
x = 200;
```

Alternatively you can `move` variables used by the closure so it owns them and they become inaccessible from the outerscope. Since our closure was accessing an integer, the move becomes an implicit copy. So our `square` closure has its own `x` assigned the value `100`. Even if we change `x` in the outer scope to `200`, the closure has its own independent copy.

```
let mut x = 100;
let square = move || x * x;
println!("square = {}", square()); // 10000
x = 200;
println!("square = {}", square()); // 10000
```

This is the equivalent to the C++ code above that used lambda expressions to bind to copies and references:

```
let mut v1 = 10.0;
let v2 = 2.0;
let multiply = move || v1 * v2;
let sum = |x: &f64, y: &f64| x + y;
println!("multiply {}", multiply());
println!("sum {}", sum(&v1, &v2));
v1 = 99.0;
println!("multiply {}", multiply());
println!("sum {}", sum(&v1, &v2));
```

This will yield the same results as the C++ code. The main difference here is that rather than binding our closure to a reference, we passed the reference values in as parameters to the closure.

Templates / Generics

C++ offers templates as a way to write generic code using an abstract type and then specialize it by substituting one or more types into a concrete class.

```
template <typename T>
inline void debug(const T &v) {
  cout << "The value of object is " << v << endl;
}
//...
debug(10);
```

This template uses the type of the parameter (int this case 10) to create an inline function that prints out the value of that type:

```
The value of object is 10
```

Classes can also be made from templates:

```
template <class T>
class Stack {
private:
  vector<T> elements;
public:
  void push(const T &v) {
    // ...
  }
  T pop() {
    // ...
  }
}
//...
Stack<double> doubleStack;
```

This class implements a simple stack using a template to indicate the type of object it contains.

This is a very powerful mechanism and the C++ library makes extensive use of it.

Where templates can become a bit of a mess is that the templates are inline and the compiler will expand out anything you call before attempting to compile it.

An innocuous error such as using a type that has no default copy constructor in a collection can cause the compiler to go nuts and output a wall of indecipherable errors.

Generic Functions

Rust's equivalent to a template is called a generic. A generic generalizes a function or a trait so it works with different types that match the criteria.

So the Rust equivalent of the `debug()` function in C++ would be this.

```
use std::fmt;

fn debug<T>(data: T) where T: fmt::Display {
  println!("The value of object is {}", data);
}
//...
debug(10);
```

Here we describe a function that takes a generic type `T` where the constraint is that `T` must implement the trait `std::fmt::Display`. Any struct that implements this trait can passed into the call. Since integer types implement the trait, we can just call it directly as `debug(10)` and the compiler is happy.

Generic structs

Similarly we can use generics on a struct. So the equivalent in Rust of the C++ template class `Stack` is this:

```
struct Stack<T> {
  elements: Vec<T>
}

impl<T> Stack<T> {
  fn new() -> Stack<T> { Stack { elements: Vec::new() } }

  fn push(v: T) {
    //...
  }

  fn pop() -> Option<T> {
    //...
    None
  }
}
//...
let double_stack: Stack<f64> = Stack::new();
```

Where clause

The `where` clause can be added to impose constraints on what generic type must do to be allowed to be supplied to the generic function or struct.

For example we might have a function that takes a closure as an argument. A closure is a function and so we want to define the shape that the closure will take.

So:

```
fn compare<T, F>(a: T, b: T, f: F) -> bool
  where F: FnOnce(T, T) -> bool
{
  f(a, b)
}

let comparer = |a, b| a < b;
let result = compare(10, 20, comparer);
```

Here we have defined a `compare()` function that takes a couple of values of the same type. The `where` clause states that the function must take two values of the same type and return a boolean. The compiler will ensure any closure we pass in matches that criteria, as indeed our `comparer` closure does.

Attributes

C++ has various ways to give compiler *directives* during compilation:

- Compile flags that control numerous behaviours
- `#pragma` statements - `once`, `optimize`, `comment`, `pack` etc. Some pragmas such as `comment` have been wildly abused in some compilers to insert "comments" into object files that control the import / export of symbols, static linking and other functionality.
- `#define` with ubquitous `#ifdef` / `#else` / `#endif` blocks
- Keywords `inline`, `const`, `volatile` etc.. These hint the code and allow the compiler to make decisions that might change its output or optimization. Compilers often have their own proprietary extensions.

Rust uses a notation called *attributes* that serves a similar role to all of these things but in a more consistent form.

An attribute `#[foo]` applies to the next item it is declared before. A common attribute is used to denote a unit test case with `#[test]`:

```
#[test]
fn this_is_a_test() {
  //...
}
```

Attributes can also be expressed as `#![foo]` which affects the thing they're contained *by* rather the thing that follows them.

```
fn this_is_a_test() {
  #![test]
  //...
}
```

Attributes are enclosed in a `#[ ]` block and provide compiler directives that allow:

- Functions to be marked as unit or benchmark tests
- Functions to be marked for conditional compilation for a target OS. A function can be defined that only compiles for one target. e.g. perhaps the code that communicates with another process on Windows and Linux is encapsulated in the same function but implemented differently.

- Enable / disable lint rules
- Enable / disable compiler features. Certain features of rust may be experimental or deprecated and may have to be enabled to be accessed.
- Change the entry point function from `main` to something else
- Conditional compilation according to target architecture, OS, family, endianess, pointer width
- Inline hinting
- Deriving certain traits
- Enabling compiler features such as plugins that implement procedural macros.
- Importing macros from other crates
- Used by certain crates like serde and rocket to instrument code - NB Rocket uses unstable compiler hooks for this and in so doing limits itself to working in nightly builds only.

Conditional compilation

Conditional compilation allows you to test the target configurations and optionally compile functions or modules in or not.

The main configurations you will test include:

- Target architecture - "x86", "x86_64", mips", "arm" etc.
- Target OS - "windows", "macos", "ios", "linux", "android", "freebsd" etc.
- Target family - "unix" or "windows"
- Target environment - "gnu", "msvc" etc
- Target endianess
- Target pointer width

So if you have a function which is implemented one way for Windows and another for Linux you might code it like so:

```
#[cfg(windows)]
fn get_app_data_dir() -> String { /* ... */ }

#[cfg(not(windows))]
fn get_app_data_dir() -> String { /* ... */ }
```

Many more possibilities are listed in the documentation.

Linking to native libraries

In C/C++ code is first compiled and then it is linked, either by additional arguments to the compiler, or by invoking a linker.

In Rust most of your linking is taken care for you providing you use `cargo`.

1. All your sources are compiled and linked together.
2. External crates are automatically built as static libs and linked in.
3. But if you have to link against something external through FFI you have to write a `#link` directive in your `lib.rs` or `main.rs`. This is somewhat analogous to the `#pragma(comment, "somelib")` in C++.

C++	Rust
`#pragma (comment, "somelib")`	`#[link(name = "somelib")]`
-	`#[link(name = "somelib", kind = "static")]`

The default kind for `#link` is `dynamic` library but `static` can be explicitly stated specified.

Inlining code

Inlining happens where your function logic is inserted in-place to the code that invokes it. It tends to happen when the function does something trivial such as return a value or execute a simple conditional. The overhead of duplicating the code is outweighed by the performance benefit.

Inlining is achieved in C++ by declaring and implementing a function, class method or template method in a header or marking it with the inline keyword.

In Rust, inlining is only a hint. Rust recommends not forcing inlning, rather leaving it as a hint for the LLVM compiler to do with as it sees fit.

C++	Rust
Explicitly with `inline` or implicitly through methods implemented in class / struct	`#[inline]`, `#[inline(always)]`, `#[inline(never)]`

Another alternative to explicitly inlining code is to use the link-time optimisation in LLVM.

```
rustc -C lto
```

Multithreading

Multithreading allows you to run parts of your programming concurrently, performing tasks in parallel. Every program has a *main* thread - i.e. the one your `main()` started from, in addition to which are any that you create.

Examples of reasons to use threads:

- Long running operations, e.g. zipping up a large file.
- Activity that is blocking in nature, e.g. listening for connections on a socket
- Processing data in parallel, e.g. physics, collision detection etc.
- Asynchronous activities, e.g. timers, polling operations.

In addition, if you use a graphical toolkit, or 3rd party libraries they may spawn their own threads that you do not know about.

Thread safety

One word you will hear a lot in multithreading is thread safety.

By that we mean:

- Threads should not be able to modify the data at the same time. When this happens it is called a data race and can corrupt the data, causing a crash. e.g. two threads trying to append to a string at the same time.
- Threads must not lock resources in a way that could cause deadlock i.e. thread 1 obtains a lock on resource B and blocks on resource A, while thread 2 obtains a lock on resource A and blocks on resource B. Both threads are locked forever waiting for a resource to release that never will be.
- Race conditions are bad, i.e. the order of thread execution produces unpredictable results on the output from the same input.
- APIs that can be called by multiple threads must either protect their data structures or make it an explicit problem of the client to sort out.
- Open files and other resources that are accessed by multiple threads must be managed safely.

Protecting shared data

Data should never be read at the same time it is written to in another thread. Nor should data be written to at the same time by two threads.

The common way to prevent this is either:

- Use a mutex to guard access to the data. A mutex is a special class that only one thread can lock at a time. Other threads that try to lock the mutex will wait until the lock held by another thread is relinquished
- Use a read-write lock. Similar to a mutex, it allows one thread to lock the thread for writing data, however it permits multiple threads to have read access, providing nothing is already writing to it. For data that is read more frequently than it is modified, this is a lot more efficient than just a mutex.

Avoiding deadlock

The best way to avoid deadlock is only ever obtain a lock to one thing ever and release it as soon as you are done. But if you have to lock more than one thing, ensure the locking order is consistent between all your threads. So if thread 1 locks A and B, then ensure that thread 2 also locks A and B in that order and not B then A. The latter is surely going to cause a deadlock.

C / C++

C and C++ predate threading to some extent so until C++11 the languages have had little built-in support for multi-threading and what there was tended to be compiler specific extensions.

A consequence of this is that C and C++ have ZERO ENFORCEMENT of thread safety. If you data race - too bad. If you forget to write a lock in one function even if you remembered all the others - too bad. You have to discipline yourself to think concurrently and apply the proper protections where it is required.

The consequence of not doing so may not even be felt until your software is in production and that one customer starts complaining that their server freezes about once a week. Good luck finding that bug!

Multithreading APIs

The most common APIs would be:

- `<thread>` , `<mutex>` - from C++11 onwards
- POSIX threads, or pthreads. Exposed by POSIX systems such as Linux and most other Unix derivatives, e.g. OS X. There is also pthread-win32 support built over the top of Win32 threads.
- Win32 threads. Exposed by the Windows operating system.

- OpenMP. Supported by many C++ compilers.
- 3rd party libraries like Boost and Qt provide wrappers that abstract the differences between thread APIs.

All APIs will have in common:

- Thread creation, destruction, joins (waiting on threads) and detaches (freeing the thread to do what it likes).
- Synchronization between threads using locks and barriers.
- Mutexes - mutual exclusion locks that protect shared data.
- Conditional variables - a means to signal and notify of conditions becoming true.

std::thread

The `std::thread` represents a single thread of execution and provides an abstraction over platform dependent ways of threading.

```
#include <iostream>
#include <thread>

using namespace std;

void DoWork(int loop_count) {
    for (int i = 0; i < loop_count; ++i) {
        cout << "Hello world " << i << endl;
    }
}

int main() {
    thread worker(DoWork, 100);
    worker.join();
}
```

The example spawns a thread which invokes the function and passes the parameter into it, printing a message 100 times.

std::mutex

C++ provides a family of various `mutex` types to protect access to shared data.

The mutex is obtained by a `lock_guard` and other attempts to obtain the mutex are blocked until the lock is relinquished.

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex data_guard;
int result = 0;

void DoWork(int loop_count) {
    for (auto i = 0; i < loop_count; ++i) {
        lock_guard<mutex> guard(data_guard);
        result += 1;
    }
}

int main() {
    thread worker1(DoWork, 100);
    thread worker2(DoWork, 150);
    worker1.join();
    worker2.join();
    cout << "result = " << result << endl;
}
```

POSIX threads

The pthreads API is prefixed `pthread_` and works like so:

```
#include <iostream>
#include <pthread.h>

using namespace std;

void *DoWork(void *data) {
    const int loop_count = (int) data;
    for (int i = 0; i < loop_count; ++i) {
        cout << "Hello world " << i << endl;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t worker_thread;
    int result = pthread_create(&worker_thread, NULL, DoWork,
(void *) 100);
    // Wait for the thread to end
    result = pthread_join(worker_thread, NULL);
}
```

This example spawns a thread which invokes DoWork with the payload of 100 which causes the function to print a message 100 times.

Win32 Threads

Win32 threading has functions analogous to those in POSIX. They have names such as `CreateThread`, `ExitThread`, `SetThreadPriority` etc.

OpenMP API

Open Multi-Processing (OpenMP) is an API for multi-threaded parallel processing. OpenMP relies on compiler support because you use special `#pragma` directives in your source to control thread creation and access to data.

GCC, Clang and Visual C++ have support for OpenMP so it is an option.

OpenMP is a complex standard but the use of directives can make for cleaner code than invoking threading APIs directly. The downside is it is also more opaque hiding what the software is doing, making it considerably more difficult to debug.

OpenMP is described in detail at the OpenMP website.

Thread local storage

Thread local storage, or TLS is static or global data which is private to every thread. Each thread holds its own copy of this data so it can modify it without fear of causing a data race.

Compilers also have proprietary ways to decorate types as thread local:

```
__thread int private; // gcc / clang
__declspec(thread) int private; // MSVC
```

C++11 has gained a `thread_local` directive to decorate variables which should use TLS.

```
thread_local int private
```

Rust

We saw with C++ that you had to be disciplined to remember to protect data from race conditions.

Rust doesn't give you that luxury -

1. Any data that you share must be protected in a thread safe fashion
2. Any data that you pass between threads must be marked thread safe

Spawning a thread

Spawning a thread is easy enough by calling `spawn`, supplying the closure you want to run in the context of your new thread.

```
use std::thread;

thread::spawn(move || {
  println!("Hello");
});
```

Alternatively you can supply a function to `spawn` which is called in the same manner.

```
fn my_thread() {
  println!("Hello");
}
//...
thread::spawn(my_thread);
```

If you supply a closure then it must have a lifetime of `'static` because threads can outlive the thing that created them. i.e. they are detached by default.

A closure can make use of move values that are marked `Send` so the compiler allows ownership to transfer between threads.

Likewise function / closure may also return a value which is marked `Send` so the compiler can transfer ownership between the terminating thread and the thread which calls `join` to obtain the value.

So the thread above is detached. If we wanted to wait for the thread to complete, the `spawn` returns a `JoinHandle` that we can call `join` to wait for termination.

```
let h = thread::spawn(move || {
  println!("Hello");
});
h.join();
```

If the closure or function returns a value, we can use `join` to obtain it.

```
let h = thread::spawn(move || 100 * 100);
let result = h.join().unwrap();
println!("Result = {}", result);
```

Data race protection in the compiler

Data races are bad news, but fortunately in Rust the compiler has your back. You MUST protect your shared data or it won't compile.

The simplest way to protect your data is to wrap the data in a mutex and provide each thread instance with a reference counted copy of the mutex.

```
let shared_data = Arc::new(Mutex::new(MySharedData::new()));

// Each thread we spawn should have a clone of this Arc
let shared_data = shared_data.clone();
thread::spawn(move || {
  let mut shared_data = shared_data.lock().unwrap();
  shared_data.counter += 1;
});
```

Here is a full example that spawns 10 threads that each increment the counter.

```
struct MySharedData {
  pub counter: u32,
}

impl MySharedData {
  pub fn new() -> MySharedData {
    MySharedData {
      counter: 0
    }
  }
}

fn main() {
  spawn_threads();
}

fn spawn_threads() {
  let shared_data = Arc::new(Mutex::new(MySharedData::new()));

  // Spawn a number of threads and collect their join handles
  let handles: Vec<JoinHandle<_>> = (0..10).map(|_| {
    let shared_data = shared_data.clone();
    thread::spawn(move || {
      let mut shared_data = shared_data.lock().unwrap();
      shared_data.counter += 1;
    })
  }).collect();

  // Wait for each thread to complete
  for h in handles {
    h.join();
  }

  // Print the data
  let shared_data = shared_data.lock().unwrap();
  println!("Total = {}", shared_data.counter);
}
```

So the basic strategy will be this:

1. Every thread will get it's own atomic reference to the mutex.
2. Each thread that wishes to access the shared must obtain a lock on the mutex.
3. Once the lock is released, the next waiting thread can obtain access.
4. The compiler will enforce this and generate errors if ANYTHING is wrong.

Read Write Lock

A read write lock works much like a mutex - we wrap the shared data in a `RwLock`, and then in an `Arc`.

```
let shared_data = Arc::new(RwLock::new(MySharedData::new()));
```

Each thread will then either need to obtain a read lock or a write lock on the shared data.

```
let shared_data = shared_data.read().unwrap();
// OR
let mut shared_data = shared_data.write().unwrap();
```

The advantage of a `RwLock` is that many threads can concurrently read the data, providing nothing is writing to it. This may be more efficient in many cases.

Sending data between threads using channels

TODO mpsc channel

Thread local storage

As with C++ you may have reason to use thread local storage

```
thread_local! {
  // TODO
}
```

Useful crates

Rayon

The rayon crate implements parallel iterators that allow your collections to be iterated in parallel. The crate utilises work stealing and divide and conquer algorithms couple to a thread pool to process collections more quickly than they could be in a sequential fashion.

Generally speaking this is a drop-in replacement with the exception that you call `par_iter` instead of `iter`. The crate implements a `ParallelIterator` trait on collection classes.

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}
```

See the crate site for more information.

Lint

C/C++ compilers can issue many useful warnings but the amount of static analysis they can do is usually quite limited.

The Rust compiler performs a far more rigorous lifecycle check on data and then follows up with a lint check that inspects your code for potentially bad or erroneous

In particular it looks for:

- Dead / unused code
- Unreachable code
- Deprecated methods
- Undocumented functions
- Camel case / snake case violations
- Unbounded recursion code (i.e. no conditionals to stop recursion)
- Use of heap memory when stack could be used
- Unused extern crates, imports, variables, attributes, mut, parentheses
- Using "while true {}" instead of "loop {}"

Lint rules can be enforced more strictly or ignored by using attributes:

```
#[allow(rule)]
#[warn(rule)]
#[deny(rule)]
#[forbid(rule)]
```

A full list of lint rules can be found by typing "rustc -W help":

```
                          name  default  meaning
                          ----  -------  -------
                  box-pointers   allow    use of owned (Box type)
heap memory
             fat-ptr-transmutes  allow    detects transmutes of
fat pointers
 missing-copy-implementations  allow    detects potentially-
forgotten implementations of `Copy`
missing-debug-implementations  allow    detects missing
implementations of fmt::Debug
                   missing-docs  allow    detects missing
documentation for public members
                  trivial-casts  allow    detects trivial casts
which could be removed
          trivial-numeric-casts  allow    detects trivial casts of
numeric types which could be removed
                    unsafe-code  allow    usage of `unsafe` code
...
```

There are a lot checks than are listed here.

Macros

C / C++ Preprocessor

C languages are little unusual in that they are compiled in two phases. The first phase is called the preprocess. In this phase, the preprocessor looks for directives starting with a # symbol and runs string substitution and conditional inclusion / exclusion based on those directives. Only after the file has been preprocessed does the compiler attempt to compile it.

Preprocessor directives start with a `#` symbol. For example the `#define` directive creates a macro with an optional value:

```
#define IS_WINDOWS
#define SHAREWARE_VERSION 1
```

We'll explore macros more in a moment. Another directive is the `#if\#else\#endif` or `#ifdef\#else\#endif` which can be used to include code from one branch or the other of a test according to what matches.

```
#if SHAREWARE_VERSION == 1
showNagwarePopup();
#endif
//...
#ifdef IS_WINDOWS
writePrefsToRegistry();
#else
writePrefsToCfg();
#endif
```

Another directive is `#include`. In C and C++, public functions and structures are typically defined and implemented in separate files. The `#include` directive allows a header to be pulled in to the front of any file that makes use of those definitions.

```
// System / external headers tend to use angle style
#include <string>
#include <stdio.h>

// Local headers tend to use double quotes
#include "MyClass.h"
```

The important thing to remember in all of this is ALL of these things happen before the compiler even starts! Your `main.c` might only be 10 lines of code but if you `#include` some headers the preprocessor may be feeding many thousands of lines of types, functions into the compiler, all of which are evaluated before they get to your code.

C / C++ Macros

Macros are string substitution patterns performed by the preprocessor before the source is compiled. As such they can be very prone to error and so have been deprecated in favour of constants and inline functions.

Here is a simple macro that would behave in an unexpected manner:

```
#define MULTIPLY(x, y) x * y
//
int x = 10, y = 20;
int result = MULTIPLY(x + 1, x + y);
// Value is NOT 330 (11 * 30), it's 41 because macro becomes x +
1 * x + y
```

The macro is very simple - multiply x by y. But it fails if either argument is an expression. Judicious use of parentheses might avoid the error in this case, but we could break it again using some pre or post increments.

Macros in C++ are also unhygenic, i.e. the macro can inadvertently conflict with or capture values from outside of itself causing errors in the code.

```
#define SWAP(x, y) int tmp = y; y = x; x = y;
//
int tmp = 10;
int a = 20, b = 30;
SWAP(a, b); // ERROR
```

Here our SWAP macro uses a temporary value called `tmp` that already existed in the scope and so the compiler complains. A macro might avoid this by using shadow variables enclosed within a `do / while(0)` block to avoid conflicts but it is less than ideal.

```
#define SWAP(x, y) do { int tmp = y; y = x; x = y } while(0);
```

Consequently inline functions are used wherever possible. Even so macros are still frequently used in these roles:

- To conditionally include for a command-line flag or directive, e.g. the compiler might `#define WIN32` so code can conditionally compile one way or another according to its presence.
- For adding guard blocks around headers to prevent them being #include'd more than once. Most compilers implement a "#pragma once directive" which is an increasingly common alternative
- For generating snippets of boiler plate code (e.g. namespace wrappers), or things that might be compiled away depending on #defines like DEBUG being set or not.
- For making strings of values and other esoteric edge cases

Writing a macro is easy, perhaps too easy:

```
#define PRINT(x) \
  printf("You printed %d", x);
```

This macro would expand to printf before compilation but it would fail to compile or print the wrong thing if x were not an integer.

Rust macros

Macros in Rust are quite a complex topic but they are more powerful and safer than the ones in C++.

- Rust macros are hygenic. That is to say if a macro contains variables, their names do not conflict with, hide, or otherwise interfere with named variables from the scope they're used from.
- The pattern supplied in between the brackets of the macro are tokenized and designated as parts of the Rust language. identifiers, expressions etc. In C / C++ you can #define a macro to be anything you like whether it is garbage or syntactically correct. Furthermore you can call it from anywhere you like because it is preprocessed even before the compiler sees the code.

- Rust macros are either declarative and rule based with each rule having a left hand side pattern "matcher" and a right hand side "substitution". Or they're procedural and actualy rust code turns an input into an output (see section below).
- Macros must produce syntactically correct code.
- Declarative macros can be exported by crates and used in other code providing the other code elects to enable macro support from the crate. This is a little messy since it must be signalled with a #[macro_export] directive.

With all that said, macros in Rust *are* complex - perhaps too complex - and generally speaking should be used as sparingly as possible.

Here is a simple declarative macro demonstrating repetition called hello_x!(). It will take a comma separated list of expressions and say hello to each one of them.

```
macro_rules! hello_x {
  ($($name:expr),*) => (
    $(println!("Hello {}", $name);)*
  )
}
// The code can supply as many arguments it likes to this macro
hello_x!("Bob", "Sue", "John", "Ellen");
```

Essentially the matcher matches against our comma separate list and the substitution generates one println!() with the message for each expression.

```
Hello Bob
Hello Sue
Hello John
Hello Ellen
```

What if we threw some other expressions into that array?

```
hello_x!("Bob", true, 1234.333, -1);
```

Well that works too:

```
Hello Bob
Hello true
Hello 1234.333
Hello -1
```

What about some illegal code:

```
hello_x!(Aardvark {});
```

We get a meaningful error originating from the macro.

```
error[E0422]: `Aardvark` does not name a structure
  |
8 | hello_x!(Aardvark {});
  |          ^^^^^^^^
<std macros>:2:27: 2:58 note: in this expansion of format_args!
<std macros>:3:1: 3:54 note: in this expansion of print!
(defined in <std macros>)
<anon>:5:7: 5:35 note: in this expansion of println! (defined in
<std macros>)
<anon>:8:1: 8:23 note: in this expansion of hello_x! (defined in
<anon>)
```

Real world example - vec!()

Rust comes with a lot of macros for reducing some of the leg work of tedious boiler plate. For example the vec!() macro is a way to declare a std::Vec and prepopulate it with some values.

Here is the actual vec! macro source code taken from the Rust source:

```
macro_rules! vec {
    ($elem:expr; $n:expr) => (
        $crate::vec::from_elem($elem, $n)
    );
    ($($x:expr),*) => (
        <[_]>::into_vec(box [$($x),*])
    );
    ($($x:expr,)*) => (vec![$($x),*])
}
```

It looks complex but we will break it down to see what it does. Firstly it has a match-like syntax with three branches that expand to anything that matches the left hand side:

First branch

The first matcher matches a pattern such as `1; 100`. The value `1` goes into `$elem`, the value `100` goes into `$n`:

```
($elem:expr; $n:expr) =>  (
        $crate::vec::from_elem($elem, $n)
    );
```

The `$crate` is a special value that resolves to the module crate which happens to be std.

So this expands to this:

```
let v = vec!(1; 100);
// 1st branch matches and it becomes this
let v = std::vec::from_elem(1, 100);
```

Second branch

The second matcher contains a glob expression - zero or more expressions separated by comma (the last comma is optional). Each matching expression ends up in `$x`:

```
($($x:expr),*) => (
        <[_]>::into_vec(box [$($x),*])
    );
```

So we can write:

```
let v = vec!(1, 2, 3, 4, 5);
// 3nd branch matches and it becomes this
let v = <[_]>::into_vec(box [1, 2, 3, 4, 5]);
```

The box keyword tells Rust to allocate the supplied array on the heap and moves the ownership by calling a helper function called into*vec() that wraps the memory array with a Vec instance. The <[*\]>:: at the front is a turbo-fish notation to make the into_vec() generic function happy.

Third branch

The third branch is a little odd and almost looks the same as the second branch. But take at look the comma. In the last branch it was next to the asterisk, this time it is inside the inner $().

```
($($x:expr,)*) => (vec![$($x),*])
```

The matcher matches when the the comma is there and if so recursively calls vec!() again to resolve to the second branch matcher:

Basically it is there so that there can be a trailing comma in our declaration and it will still generate the same code.

```
// 3rd branch matches this
let v = vec!(1, 2, 3, 4, 5,);
// and it becomes this
let v = vec!(1, 2, 3, 4, 5);
// which matches 2nd branch to become
let v = <[_]>::into_vec(box [1, 2, 3, 4, 5]);
```

Procedural Macros

So far we've talked about declarative macros that expand out to be Rust code based upon how they pattern match the rules defined by the macro.

A second kind of macro is the *procedural macro*. A procedural macro is a plugin written in Rust that is compiled and loaded by the compiler to produce arbitrary Rust code as its output.

A procedural macro can therefore be thought of as a code generator but one that forms part of the actual compiler. Procedural macros can be particularly useful for:

- Serialization / deserialization (e.g. the serde module generates code for reading and writing structs to a variety of formats - JSON, YAML, TOML, XML etc.)
- Domain Specific Languages (e.g. embedded SQL, regular expressions etc).
- Aspect oriented programming (e.g. extra debugging, performance metrics etc)
- New lint and derive rules

For more information look at this section on compiler plugins in the Rust book.

Other forms of conditional compilation

We saw that the C / C++ preprocessor can be used for conditional compilation. The equivalent in Rust is attributes. See the attributes section to see how they may be used.

Memory allocation

This section is concerned with memory allocation, i.e. creating objects that reside on the heap and not on the stack, and the manner in which they are created and are destroyed.

C++

C and C++ have various standard ways to allocate memory:

1. `malloc` / `calloc` / `realloc()` and `free()` functions
2. `new` and `delete` (C++ only)
3. `new[]` and `delete[]` for arrays (C++ only)

Invoking `malloc()` / `free()` on a C++ class or struct is never a good idea since it will not call the corresponding class constructor or destructor. The `realloc()` function allocates a new piece of memory, copying the contents of an existing piece of memory before freeing the original.

```
// malloc / free
char *buffer = (char *) malloc(1024);
...
free(buffer);
// new / delete
Stack *stack = new Stack();
...
delete stack;
// new[] / delete[]
Node *nodes = new Node[100];
...
delete []nodes;
```

In each case the allocation must be matched by the corresponding free action so immediately we can see scope for error here:

1. Ownership rules can get messy, especially when a class is passed around a lot - who deletes the object and when?
2. Not using the correct `new` & `delete` pair, causing a memory leak. e.g. calling `delete` instead of `delete[]`

3. Forgetting to free memory at all causing a memory leak.
4. Freeing memory more than once.
5. Calling a dangling pointer, i.e. a pointer which refers to freed memory.
6. Allocating / freeing in a way that causes heap fragmentation. Reallocation can cause fragmentation to happen a lot faster.

C++ has smart pointers which manage the lifetime on objects and are a good way to programmer error:

```
{
  std::auto_ptr<Database> db(new Database());
  //... object is deleted when db goes out of scope
}

// C++11
{
  std::unique_ptr<Database> db(new Database());
  //... object is deleted when db goes out of scope

  std::unique_ptr<Node[]> nodes<new Node[100]);
  //... arrays of objects are supported too
}

// C++11
{
  std::shared_ptr<Database> db(new Database());
  // Reference count db
  setDatabase(db);
  //... object is deleted when last shared_ptr reference to it
goes out of scope

  std::shared_ptr<Node[]> nodes<new Node[100]);
  //... arrays of objects are supported too
}
```

Unfortunately it is not always possible to use smart pointers but wherever possible they should be used.

Other ways of allocating memory

Virtually every C and C++ library has solutions for managing memory. They all their own indivual concept of ownership which is usually different from one to the next. Boost and Qt have their own memory management "smart" pointers. Qt even requires certain objects to be deleted "later" by a message processing loop on the thread that created the object. Some libraries even adopt a COM-like model of reference counting objects with smart pointers. Most C libraries will expose an alloc and free function for creating and destroying context objects that callers pass to the API.

Memory allocation can even be overwritten and replaced in some circumstances. In C, the standard malloc / free can be substituted for another memory allocator, e.g. TCMalloc TCMalloc. Or perhaps the code wants to use garbage collected memory in which case Bohem GC is a popular library for that purpose. Boehm can also be used for leak detection since it can find objects which were never released. C++ can also override the global or class specific new / delete operators. Some standard C++ template classes also allow memory allocation to be overridden.

Rust

As you can guess by now Rust tends to be a lot more strict about allocation that C/C++. Lifetimes of objects are tracked and enforced by the compiler and that includes memory allocated objects.

In normal safe programming there is no explicit new / delete so there is no way to forget to free an object. There are no pointers either so code cannot call a dangling pointer or inadvertently call a null pointer.

1. A `Box` is a managed pointer that holds a heap allocated object. A box cannot be cloned, so there is only one owner at any time.
2. A `Cell` is a mutable memory location - it can hold any kind of copyable type and the value within it can be changed.
3. A `RefCell` is a mutable memory location that can hold a reference

The advantage for programmers, is that once you define the lifetime of an object properly it just comes into existence and goes away correctly. In many cases this lifetime management comes with zero runtime cost, or if there is a cost it is no more than the same code correctly written in C/C++.

Rust requires most heap allocated memory to be contained by one or more of the structs below. The struct manages the lifetime and access to the object inside ensuring the lifetime is managed correctly.

Box

A `Box` is memory managed on the heap.

```
struct Blob {
  data: Box<[u8; 16384]>
}

impl Blob {
  pub fn new() {
    Efficient {
      data: Box::new([0u8; 16384])
    }
  }
}
```

Whoever owns the box can access it. Essentially, that means you can pass the box around from one place to another and whatever binds to it last can open it. Everyone else's binding becomes invalid and will generate a compile error.

A box can be useful for abstraction since it can refer to a struct by a trait it implements allowing decoupling between types.

TODO example of a struct holding a box with a trait implemented by another struct

It can be useful for situations where one piece of code creates an object on behalf of another piece of code and hands it over. The Box makes sure that the ownership is explicit at all times and when the box moves to its new owner, so does the lifetime of the object itself.

Cell

A `Cell` is something that can copied with a `get()` or `set()` to overwrite its own copy. As the contents must be copyable they must implement the Copy trait.

The `Cell` has a zero-cost at runtime because it doesn't have to track borrows but the restriction is it only works on Copy types. Therefore it would not be suitable for large objects or deep-copy objects.

RefCell

Somewhat more useful is the `RefCell<T>` but it incurs a runtime penalty to maintain read-write locks.

The `RefCell` holds a reference to an object that can be borrowed either mutably or immutably. These references are read-write locked so there is a runtime cost to this since the borrow must check if something else has already borrowed the reference.

Typically a piece of code might borrow the reference for a scope and then the borrow disappears when it goes out of scope. If a borrow happens before the last borrow releases, it will cause a panic.

Reference Counting objects

Rust implements `Rc<>` and `Arc<>` for the purpose of reference counting objects that need to be shared and used by different parts of code. Rc<> is a single threaded reference counted wrapper, while `Arc<>` is atomic reference counted wrapper. You use one or the other depending on whether threads are sharing the object.

A reference counted object is usually wrapping a `Box` , `Cell` or `Refcell` . So multiple structs can hold a reference to the same object.

Rc

From `std::rc::Rc` . A reference counted object can be held by multiple owners at a time. Each own holds a cloned `Rc<T>` but the T contents are shared. The last reference to the object causes the contents to be destroyed.

Arc

From `std::sync::Arc` . An atomic reference counted object that works like `Rc<T>` except it uses an atomically incremented counter which makes it thread safe. There is more overhead to maintain an atomic reference count. If multiple threads access the same object they are compelled to use `Arc<T>`

Foreign Function Interface

Rust doesn't work in a vaccum and was never intended as such. Instead it was always assumed that it would need to call other code and other code would need to call it,

- Call other libraries via their entry points
- Produce libraries in Rust that can be called by code written in another language. e.g. C, C++, Python, Ruby etc.

To that end it has the Foreign Function Interface, the means to define external functions, expose its own functions without name mangling and to invoke unsafe code that would otherwise be illegal in Rust.

Calling out to C libraries

Rust supports the concept of a foreign function interface which is a definition of an external function or type that is resolved at link time.

For example, we might wish to link to a library called foo.lib, and invoke a command foo_command().

```
#[link(name = "foo")]
extern {
  fn foo_command(command: *mut u8)
}
```

To call this function we have to turn off safety checks first because we are stepping out of the bounds of Rust's lifetime enforcement. To do this we wrap the call in an unsafe block to disable the safety checks:

```
pub fn run_command(command: &[u8]) {
  unsafe {
    foo_command(command.as_ptr());
  }
}
```

Note how we can use unsafe features like pointers inside of this unsafe block. This allows interaction with the outside world while still enforcing safety for the rest of our code.

Making Rust code callable

The converse is also possible. We can produce a library from Rust that can be invoked by some other code.

For example, imagine we have some code written in Python. The code works fine but it is not performant and the bottle neck is in just one portion of the code, e.g. some file operation like a checksum. We want our code to consist of a make_checksum() and a release_checksum().

```
extern crate libc;

use std::ffi::CString;
use std::ptr;
use libc::{c_char, c_void, malloc, memset, strcpy, free};

#[no_mangle]
pub extern "C" fn make_checksum(filepath: *const c_char) -> *mut
c_char {
    // Your code here
    if filepath == ptr::null() {
      return ptr::null_mut::<c_char>()
    }

    unsafe {
        // Imagine our checksum code here...
        let result = malloc(12);
        memset(result, 0, 12);
        strcpy(result as *mut c_char,
CString::new("abcdef").unwrap().as_ptr());
        return result as *mut c_char;
    }
}

#[no_mangle]
pub extern "C" fn release_checksum(checksum: *const c_char) {
    unsafe {
        free(checksum as *mut c_void);
    }
}
```

Now in Python we can invoke the library simply:

```
import ctypes

checksum = ctypes.CDLL("path/to/our/dll");
cs = checksum.make_checksum("c:/somefile");
...
checksum.release_checksum(cs)
```

The FFI specification goes into a lot more detail than this and explains concepts such as callbacks, structure packing, stdcall, linking and other issues that allow full interoperability.

libc

Rust maintains a crate called libc which holds types and functions corresponding to C.

A dependency to libc would be added to the `Cargo.toml` of your project:

```
[dependencies]
libc = "0.2.17"
```

And the file that uses the functions would contain a preamble such as this saying what types and functions it calls:

```
extern crate libc;

use libc::{c_char, malloc, free, atoi};
```

Other libraries

There are also crates that have the definitions of structures, types and functions.

- WinAPI bindings for Win32 programming APIs.
- OpenSSL bindings for OpenSSL

Fixing Problems in C/C++

This section is not so much concerned with *the correct* way code should be written but what with the C/C++ languages allow. Successive versions of these languages have attempted to retrofit good practice, but not by eliminating the bad one!

All of these things should be considered bad and any construct in the language that enables them is also bad:

- Calling a pure virtual function in a constructor / destructor of a base class
- Calling a dangling pointer
- Freeing memory more than once
- Using default copy constructors or assignment operators without following the rule of three
- Overflowing a buffer, e.g. being off by one with some string operation or not testing a boundary condition
- Memory leaks due to memory allocation / ownership issues
- Heap corruption

The C++ programming language is a very large specification, one that only grows and gets more nuanced and qualified with each release.

The problem from a programmer's perspective is understanding what things C++ allows them to do as oppose to what things they should do.

In each case we'll see how Rust might have stopped us getting into this situation in the first place.

What about C?

C++ will come in for most of the criticism in this section. Someone might be inclined to think that therefore C does not suffer from problems.

Yes that is true to some extent, but it is akin to arguing we don't need shoes because we have no legs. C++ exists and is popular because it is perceived as a step up from C.

- Namespaces
- Improved type checking
- Classes and inheritance
- Exception handling
- More useful runtime library including collections, managed pointers, file io etc.

The ability to model classes and bind methods to them is a major advance. The ability to write RAII style code does improve the software's chances of keeping its memory and resource use under control.

Compilers Will Catch Some Errors

Modern C/C++ compilers can spot some of the errors mentioned in this section. But usually they'll just throw a warning out. Large code bases always generate warnings, many of which are innocuous and it's easy to see why some people become numb to them as they scroll past.

The simplest way to protect C / C++ from dumb errors is to elevate serious warnings to be errors. While it is not going to protect against every error it is still better than nothing.

- In Microsoft VC++ enable a high warning level, e.g. /W4 and possibly /WX to warnings into errors.
- In GCC enable -Wall, -pedantic-errors and possibly -Werror to turn warnings into errors. The pedantic flag rejects code that doesn't follow ISO C and C++ standards. There are a lot of errors that can be configured.

However this will probably throw up a lot of noise in your compilation process and some of these errors may be beyond your means to control.

In addition it is a good to run a source code analysis tool or linter. However these tend to be expensive and in many cases can be extremely unwieldy.

Copy Constructor / Assignment Operators

In C++ you can construct one instance from another via a constructor and also by an assignment operator. In some cases a constructor will be used instead of an assignment:

```
PersonList x;
PersonList y = x; // Copy constructor, not assignment
PersonList z;
z = x; // Assignment operator
```

By default C++ generates all the code to copy and assign the bytes in one class to another without any effort. Lucky us!

So our class PersonList might look like this:

```
struct Person {
    //...
};

class PersonList {
  std::vector<Person> *personList_;
public:
  PersonList() : personList_(new std::vector<Person>) {
  }

  ~PersonList() {
    delete personList_;
  }

  // ... Methods to add / search list
};
```

Except we're not lucky, we just got slimed. The default byte copy takes the pointer in `personList_` and makes a copy of it. Now if we copy `x` to `y`, or assign `x` to `z` we have three classes pointing to the same private data! On top of that, `z` allocated its own `personList_` during its default constructor but the byte copy assignment overwrote it with the one from `x` so its old `personList_` value just leaks.

Of course we might be able to use a `std::unique_ptr` to hold our pointer. In which case the compiler would generate an error. But it might not always be that simple. `personList_` may have been opaquely allocated by an external library so have no choice but to manage its lifetime through the constructor and destructor.

The Rule of Three

This is such a terrible bug enabling problem in C++ that it has given rise to the so-called the Rule of Three[1].

The rule says that if we explicitly declare a destructor, copy constructor or copy assignment operator in a C++ class then we probably need to implement all three of them to safely handle assignment and construction. In other words the burden for fixing C++'s default and dangerous behaviour falls onto the developer.

So let's fix the class:

```
struct Person {
    //...
};

class PersonList {
    std::vector<Person> *personList_;
public:
    PersonList() : personList_(new std::vector<Person>) {
    }

    PersonList(const PersonList &other) :
            personList_(new std::vector<Person>)    {
        personList_->insert(
                personList_->end(), other.personList_->begin(),
                other.personList_->end());
    }

    ~PersonList() {
        delete personList_;
    }

    PersonList & operator=(const PersonList &other) {
        // Don't forget to check if someone assigns an object to
itself
        if (&other != this) {
            personList_->clear();
            personList_->insert(
                    personList_->end(), other.personList_-
>begin(),
                    other.personList_->end());
        }
        return *this;
    }

    // ... Methods to add / search list
};
```

What a mess!

We've added a copy constructor and an assignment operator to the class to handle copying safely. The code even had to check if it was being assigned to itself in case someone wrote `x = x`. Without that test, the receiving instance would clear itself in preparation to adding elements from itself which would of course wipe out all its contents.

Alternatively we might disable copy / assignments by creating private constructors that prevents them being called by external code:

```
class PersonList {
    std::vector<Person> *personList_;

private:
    PersonList(const PersonList &other) {}
    PersonList & operator=(const PersonList &other) { return
*this; }

public:
    PersonList() : personList_(new std::vector<Person>) {
    }

    ~PersonList() {
        delete personList_;
    }
    // ... Methods to add / search list
};
```

Another alternative would be to use noncopyable types within the class itself. For example, the copy would fail if the pointer were managed with a C++11 `std::unique_ptr` (or Boost's `boost::scoped_ptr`).

Boost also provides a `boost::noncopyable` class which provides yet another option. Classes may inherit from noncopyable which implements a private copy constructor and assignment operator so any code that tries to copy will generate a compile error.

The Rule of Five

The Rule of Three has become the Rule of Five(!) in C++11 because of the introduction of move semantics.

If you have a class that can benefit from move semantics, the Rule of Five essentially says that the existence of the user-defined destructor, copy constructor and copy assignment operator requires you to also implement a move constructor and a move assignment operator. So in addition to the code we wrote above we must also write two more methods.

```
class PersonList {
  // See class above for other methods, rule of three....

  PersonList(PersonList &&other) {
    // TODO
  }

  PersonList &operator=(PersonList &&other) {
    if (&other != this) {
      // TODO
    }
    return  *this
  }
```

How Rust helps

Move is the default

Rust helps by making move semantics the default. i.e. unless you need to copy data from one instance to another, you don't. If you assign a struct from one variable to another, ownership moves with it. The old variable is marked invalid by the compiler and it is an error to access it.

But if you do want to copy data from one instance to another then you have two choices.

- Implement the `Clone` trait. Your struct will have an explicit `clone()` function you can call to make a copy of the data.
- Implement the `Copy` trait. Your struct will now implicitly copy on assignment instead of move. Implementing `Copy` also implies implementing `Clone` so you can still explicitly call `clone()` if you prefer.

Primitive types such as integers, chars, bools etc. implement `Copy` so you can just assign one to another

```
// This is all good
let x = 8;
let y = x;
y = 20;
assert_eq!(x, 8);
```

But a `String` cannot be copied this way. A string has an internal heap allocated pointer so copying is a more expensive operation. So `String` only implements the `Clone` trait which requires you to explicitly duplicate it:

```
let copyright = "Copyright 2017 Acme Factory".to_string();
let copyright2 = copyright.clone();
```

The default for any struct is that it can neither be copied nor cloned.

```
struct Person {
  name: String,
  age: u8
}
```

The following code will create a `Person` object, assigns it to `person1`. And when `person1` is assigned to `person2`, ownership of the data also moves:

```
let person1 = Person { name: "Tony".to_string(), age: 38u8 };
let person2 = person1;
```

Attempting to use `person1` after ownership moves to `person2` will generate a compile error:

```
println!("{}", person1.name); // Error, use of a moved value
```

To illustrate consider this Rust which is equivalent to the PersonList we saw in C++

```
struct PersonList {
    pub persons: Vec<Person>,
}
```

We can see that `PersonList` has a `Vec` vector of `Person` objects. Under the covers the `Vec` will allocate space in the heap to store its data.

Now let's use it.

```
let mut x = PersonList { persons: Vec::new(), };
let mut y = x;
// x is not the owner any more...
x.persons.push(Person{ name: "Fred".to_string(), age: 30u8} );
```

The variable `x` is on the stack and is a `PersonList` but the persons member is partly allocated from the heap.

The variable `x` is bound to a PersonList on the stack. The vector is created in the heap. If we assign `x` to `y` then we could have two stack objects sharing the same pointer on the heap in the same way we did in C++.

But Rust stops that from happening. When we assign `x` to `y`, the compiler will do a bitwise copy of the data in x, but it will bind ownership to `y`. When we try to access the in the old var Rust generates a compile error.

```
error[E0382]: use of moved value: `*x.persons`
   |
10 | let mut y = x;
   |     ----- value moved here
11 | x.persons.push(Person{});
   | ^^^^^^^^^ value used here after move
   |
   = note: move occurs because `x` has type `main::PersonList`,
which does not implement the `Copy` trait
```

Rust has stopped the problem that we saw in C++. Not only stopped it but told us why it stopped it - the value moved from x to y and so we can't use x any more.

Implementing the Copy trait

The `Copy` trait allows us to do direct assignment between variables. The trait has no functions, and acts as a marker in the code to denote data that should be duplicated on assignment.

You can implement the `Copy` trait by deriving it, or implementing it. But you can only do so if all the members of the struct also derive the trait:

```
#[derive(Copy)]
struct PersonKey {
  id: u32,
  age: u8,
}

// Alternatively...

impl Copy for PersonKey {}

impl Clone for PersonKey {
  fn clone(&self) -> PersonKey {
      *self
  }
}
```

So `PersonKey` is copyable because types `u32` and `u8` are also copyable and the compiler will take the `#[derive(Copy)]` directive and modify the move / copy semantics for the struct.

But when a struct contains a a type that does not implement `Copy` you will get a compiler error. So this struct `Person` will cause a compiler error because `String` does not implement `Copy:`

```
#[derive(Copy)]
struct Person {
  name: String,
  age: u8
}
// Compiler error!
```

Implementing the Clone trait

The `Clone` trait adds a `clone()` function to your struct that produces an independent copy of it. We can derive it if every member of the struct can be cloned which in the case of `Person` it can:

```
#[derive(Clone)]
struct Person {
  name: String,
  age: u8
}
...
let x = Person { /*...*/ };
let y = x.clone();
```

Now that Person derives `Clone`, we can do the same for PersonList because all its member types implement that trait - a Person can be cloned, a Vec can be cloned, and a Box can be cloned:

```
#[derive(Clone)]
struct PersonList {
    pub persons: Box<Vec<Person>>,
}
```

And now we can clone `x` into `y` and we have two independent copies.

```
//...
let mut x = PersonList { persons: Box::new(Vec::new()), };
let mut y = x.clone();
// x and y are two independent lists now, not shared
x.persons.push(Person{ name: "Fred".to_string(), age: 30} );
y.persons.push(Person{ name: "Mary".to_string(), age: 24} );
```

Summary

In summary, Rust stops us from getting into trouble by treated assigns as moves when a non-copyable variable is assigned from one to another. But if we want to be able to clone / copy we can make our intent explicit and do that too.

C++ just lets us dig a hole and fills the dirt in on top of us.

Missing Braces in Conditionals

Every programmer eventually encounters an error like this and spends hours trying to figure out why it wasn't working.

```
const bool result = fetch_files();
if (result) {
  process_files()
}
else
  print_error()
  return false;

// Now cleanup and return success
cleanup_files();
return true;
```

The reason of course was the else statement wasn't enclosed in braces so the wrong code was executed. The compiler might spot dead code in this instance but that may not always be the case. Even if it did, it might only issue a warning instead of an error.

The problem can be especially annoying in deeply nested conditions where a misplaced brace can attach to the wrong level. This problem has lead real-world security issues. For example here is the infamous "goto fail" bug that occured in some Apple products. This (intentional?) bug occured during an SSL handshake and was exploitable. :

```
static OSStatus
SSLVerifySignedServerKeyExchange(
   SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
   uint8_t *signature, UInt16 signatureLen)
{
  OSStatus        err;
  //...

  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
  //...

fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
}
```

Note how the "goto fail" is repeated twice and not bound to the condition but is indented as if it was. The code would jump straight into the fail label and return with an err indicating success (since the prior SHA1 update had succeeded). If conditionals

How Rust helps

Rust requires if-else expressions and loops to be associated with blocks.

So this code won't compile:

```
let mut x: i32 = do_something();
if x == 200 {
  // ...
}
else
  println!("Error");
```

If you try you will get an error like this.

```
rustc 1.13.0-beta.1 (cbbeba430 2016-09-28)
error: expected `{`, found `println`
  |
8 |   println!("Error");
  |   ^^^^^^^
  |
help: try placing this code inside a block
  |
8 |   println!("Error");
  |   ^^^^^^^^^^^^^^^^^^
error[E0425]: unresolved name `do_something`
  |
3 | let mut x: i32 = do_something();
  |                  ^^^^^^^^^^^^ unresolved name
```

Assignment in Conditionals

The omission of an `=` in an `==` condition turns it into an assignment that evaluates to true:

```
int result = getResponseCode();
if (result = 200) { // BUG!
  // Success
}
else {
  //... Process error
}
```

So here, result was assigned the value 200 rather than compared to the value 200. Compilers should issue a warning for these cases, but an error would be better.

Developers might also try to reverse the left and right hand side to mitigate the issue:

```
if (200 = result) { // Compiler error
  // Success
}
else {
  // ... Process error
}
```

Now the compiler will complain because the value of result is being assigned to a constant which makes no sense. This may work if a variable is compared to a constant but arguably it makes the code less readable and wouldn't help if the left and right hand sides were both assignable so their order didn't matter.

The `goto fail` example that we saw in section "Missing braces in conditionals" also demonstrates a real world dangers combining assignment and comparison into a single line:

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
  goto fail;
```

This line is not broken for other reasons, but it's easy to see how might be, especially if this pattern were repeated all over the place. The programmer might have saved a few lines of code to combine everything in this way but at a greater risk. In this case, the risk might be

inadvertantly turning the `=` into an `==`, i.e. comparing err to the function call and then comparing that to 0.

```
if ((err == SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
  goto fail;
```

How Rust helps

This code just won't compile:

```
let mut result = 0;
if result = 200 { // Compile Error
  //...
}
```

The only form of assignment inside a conditional is the specialised and explicit `if let` and `while let` forms which are explained elsewhere.

Class Member Initialisation

C++ does not require that you initialise all variables in every constructor.

- A member that is a C++ class with its own default constructor doesn't need to be initialised
- A member that is a C++ class without a default constructor must be explicitly initialised.
- A member that is a reference must be explicitly initialised
- Primitive types, including pointers do not have to be initialised although the compiler may warn if they are not
- Members do not have to be initialised in the order they are declared

Some compilers may issue warnings if you forget to initialise members or their ordering, but they will still compile the code.

C++11 allows classes to have default member initializers which are used in the absence of a constructor setting the value to something else:

```
class Coords {
public:
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;

    // 2D initializer, x and y are set with the inputs, z is set
to 0
    Coords(double x, double y) : x(x), y(y) {}
};
```

This is obviously a lot easier to read and ensures that if we have multiple constructors that we don't have to initialize members if the default value will do.

How Rust helps

You MUST initialise all members of a struct. If your code does not initialise a struct you will get a compiler error.

This will not compile:

```
struct Alphabet {
  a: i32,
  b: u32,
  c: bool,
}

let a = Alphabet { a: -10, c: true };
```

If you try you will get an error like this:

```
rustc 1.13.0-beta.1 (cbbeba430 2016-09-28)
error[E0063]: missing field `b` in initializer of
`main::Alphabet`
  |
9 |     let a = Alphabet { a: -10, c: true };
  |             ^^^^^^^^ missing `b`
```

Forcing you to initialise the members of the struct ensures the struct is always in a consistent predictable state.

Ordering of initialisation does not matter providing all of the fields are set.

Structs often implement a `new()` function which encapsulates this initialisation and acts like a constructor in C++, e.g.

```
struct Coord {
  pub x: f64,
  pub y: f64,
  pub z: f64,
}

impl Coord {
  pub fn new(x: f64, y:f64) {
    Coord { x: x, y: y, z: 0f64 }
  }
}
///...
let coord1 = Coord::new(100f64, 200f64);
```

Alternatively the struct might implement one or more `From<>` traits:

```
impl From<(f64, f64)> for Coord {
  fn from(value: (f64, f64)) -> Coord {
    Coord { x: value.0, y: value.1, z: 0.0 }
  }
}

impl From<(f64, f64, f64)> for Coord {
  fn from(value: (f64, f64, f64)) -> Coord {
    Coord { x: value.0, y: value.1, z: value.2 }
  }
}


//...
let coord = Coord::from((10.0, 20.0));
let coord = Coord::from((10.0, 20.0, 30.0));
```

There can be multiple `From` trait implementations so we can implement a form of polymorphism.

Headers and Sources

A header file contains definitions of classes, types, macros etc that other files need to #include in order to resolve their use of those things.

Splitting the implementation and definition across different files is an added burden for maintaining code but it can also lead to some serious errors.

- Headers used across multiple projects that have different compiler settings
- Issues with pragmas and alignment
- Issues with different #definitions that affect byte length
- Issues with different typedefs that affect byte length

Each consumer of the header must do so with the exact same settings that affect the size of every type, struct and class in the file plus any issues with packing / alignment. If these settings are not the same, it can cause instability, corruption or problems that only manifest themselves at at runtime.

Headers also make the compiler slower because source that consumes the header inevitably pulls in other headers which pull in other headers.

Guard blocks / #pragma once

Headers will also be expanded as many times as they are `#include`'d. To prevent the expansion happening more than once per source file, they're usually protected by guard blocks.

```
#ifndef FOO_H
#define FOO_H
....
#endif
```

If the same header is included more than once, the second time through it is preprocessed into nothing.

#pragma once

Most modern compilers also support a `#pragma once` directive. This allows the compiler to completely ignore an `#include` which it knows it has already included at least once before per source file.

This is more efficient than guard blocks because the compile doesn't even bother opening or processing the file again and just skips over it. There may be situations where this is not suitable, but usually it results in faster compilation.

Precompiled Headers

Some compilers also support precompiled headers to speed up compilation. The compiler builds a database lookup when compiling a single source file and subsequent source compiles with reference to that database. This solution can speed up compilation but it complicates the build process since one file has flags to generate the precompiled header file and other sources have flags to reference it.

Pimpl pattern

A popular workaround for header issues is the Pimpl pattern. It is a way to separate a class into a public part and a private implementation part.

The public class is almost an interface definition in its purity that can be defined in the header with minimal dependencies. It forward references the implementation class and stores it as a member:

```
#pragma once

// Gory details are in the .cpp file
class ComplexThingImpl;

class ComplexThing {
  ComplexThingImpl *pimpl_;
public:
  ComplexThing();
  ~ComplexThing();

  // See note 1 below

  void somethingReallyComplex();
};
```

The constructor for the outer class would allocate the implementation class and method calls would call through to the inner.

The private implementation class is defined in the source file and can pull in as many extra headers as it needs, pragmas whatever without hurting consumers or compile times of the header.

```
// source file
#include "random/header.hpp"
// Lots of includes here
#include <...>
#include "more/stuff.hpp"

class  ComplexThingImpl {
  // Lots of member variables and stuff here
  // ...
public:
  void somethingReallyComplex();
}

void ComplexThingImpl::somethingReallyComplex() {
  // Lots of complex stuff here
  // ...
}

ComplexThing::ComplexThing() :
  pimpl_(new ComplexThingImpl()) {
}

ComplexThing::~ComplexThing() {
  delete pimpl_;
}

void ComplexThing:: somethingReallyComplex() {
  pimpl_->somethingReallyComplex();
}
```

This solution is known as Pimpl (private implementation) pattern and while it can work to protect consumers and speed up builds it also adds complexity and overhead to development. Instead of 2 definitions of a class to maintain (header / source) you now have

4(!) because there is a public and private impl class. Changing the signature of a method means changing it in potentially 4 places, plus the line in the public class that invokes the private counterpart.

One danger for Pimpl is that the private class is allocated from the heap. Code that uses a lot of temporary Pimpl objects could contribute to heap fragmentation.

Note 1: Remember the rule of three? That applies to this object too. The example doesn't show it but if we copy constructed or assigned ComplexThing to another instance we'd be in a heap of trouble. So on top of the issues with making PImpl work we also have to prevent the other ones. The easiest way to lock it down would be to derive from `boost::noncopyable` if you were using boost or make the copy constructor `private` , or use delete it in C++11.

How Rust helps

In Rust the definition and the implementation are the same thing. So immediately we have exactly one thing to maintain.

Writing a function defines the function. Let's assume we have a functions.rs file

```
// functions.rs
pub fn create_directory_structure() {
  // Implementation
}
```

Anyone can call it as `functions::create_directory_structure()` . The compiler will validate the call is correct.

A struct's definition and its implementation are also written once. e.g. `directory.rs`

```
// directory.rs
pub struct Directory {
  pub path: String,
}
impl Directory {
  pub fn mkdir(&self) {
    // implementation
  }
}
```

Implementations can be defined in a private Rust module and only public structs exposed to consumers.

If we were a library crate (which we'll call `file_utils`) wishing to expose these objects to consumers we would write a top-level `lib.rs` which says what files our lib comprises of and we want to expose.

```
// lib.rs for file_utils
mod functions;
mod directory;
pub use functions::*;
pub use directory::Directory;
```

Now a consumer can use our crate easily:

```
extern crate file_utils;
use file_utils::*;
fn main() {
   create_directory_structure();
   let d = Directory { /* ... */ };
}
```

Forward Declarations

C++ prevents us from referring to a class or function which has not been defined yet. The compiler will complain even if the class or function is in the same file it is referenced from.

This means ordering matters. If our function or class is used by other files, we have to declare the function in a header. If our function is private to a source file, we have to declare it in the source file, and possibly make it static.

For classes we can make a forward reference. This acts as a hint to compiler to say a class does exist with this name and it will be told about it shortly. But it's a hack and it imposes limits on how we can use the forward declared class.

For example, DataManager below can hand out Data objects but the Data object has a reference to the DataManager. Since each class refers to each other there is no simple way to make the compiler happy except with a forward declaration.

```
class Data; // Forward declaration

class DataManager {
public:
  Data *getDataById(const std::string &id);
};

class Data {
public:
  Data(DataManager &dataManager);
}
```

But forward declaration compromises the design of the code. For example we couldn't hold the Data objects in a collection class:

```
class Data;

class DataManager {
  std::map<std::string, Data> data_;
public:
  Data *getDataById(const std::string &id);
}
```

The compiler would complain because it doesn't know anything about the constructors or size of Data. So instantly the design has to change because of a dumb compiler restriction. e.g. we might store a pointer to Data instead in the map but then we'd have to remember to delete it. So forward references increase the potential for bugs.

```
class Data;

class DataManager {
  // Great, now we have to remember to new / delete Data and we
increase
  // memory fragmentation
  std::map<std::string, Data*> data_;
public:
  Data *getDataById(const std::string &id);
}
```

How Rust helps

In Rust forward declarations are unnecessary. The struct and function’s definition reside in a .rs and can be referenced with a use directive.

Namespace Collisions

C code has no namespaces at all and namespaces in C++ are optional.

- C has learned to live without namespaces. Most C code uses prefixes on functions and structs to avoid collisions, e.g `sqlite3_exec()` is a function belonging to SQLite3. The prefix stops the function colliding with `exec()` which is a standard POSIX function that got there first. So the prefix acts as a pseudo namespace. But it adds noise to our code and would not be necessary if namespaces were supported and enforced.
- C++ makes them easy to declare but there is no compunction for any code to bother or to do so in anything but the most perfunctory way.
- Macros are not affected by namespaces. For example, if `TRUE` and `FALSE` are defined by some header they taint everything that `#include` 's those definitions.

By default all C++ code resides in a global namespace:

```
void hello() {
  // My function hello is in the global namespace, i.e.
::hello()
}

int main() {
  // Main entry point
  hello();
}
```

The function `hello()` is part of the global namespace. The call to it within `main` could be replaced with calls to `::hello()` . The problem of course is that the more code we write into the global namespace, or the more libraries we pull in that have no namespaces, the more chance there is of collisions.

Namespacing requires code enclose the namespaced portion in a block.

```
namespace application {
  // stuff in here belongs to application::
}
//...
application::App app("my app");
```

It is also easy to abuse namespaces, for example this happens sometimes and is NOT a good idea:

```
// Inside of foo.h...
using namespace std;
//... all code after here is tainted with std
```

Any file that says `#include "foo.h"` will inadvertently tell the compiler to automatically look up unscoped types and functions against std which may not be what the code wants at all.

Nested namespacing is also possible but it can look messy.

```
namespace application { namespace gui {
  // stuff in here belongs to application::gui::
} }
//... eg.
application::gui::Point2d point(100,100);
```

If we forget to close a brace when nesting headers it becomes very easy to make C++ throw up a wall of incoherent errors.

How Rust helps

In Rust every file is implicitly a module (equivalent to a namespace). You cannot NOT use modules because you get them automatically.

If you have a collision between the names of crates or modules y

Macros

Macros in C/C++ are basically little rules that are defined by a preprocessor and substituted into the code that the compiler ultimately attempts to compile.

Modern coding practice these days is to use inline functions and constants instead of macros.

But the reality is they can still be (ab)used and code often does. For example code might insert debug statements or logging which is compiled away in release mode.

Another common use is on Windows where the type `TCHAR` compiles to be either `char` or `wchar_t` depending on `#define UNICODE` being present or not. Along with it go macros like `USES_CONVERSION` , `A2CT` , `T2CW` etc. Code should compile cleanly either way but the reality is usually it doesn't.

A classic problem would be something like this:

```
#define SQUARED(x) x * x
// And in code
float result = SQUARED(++x);
That would expand to
float result = ++x * ++x;
```

So the value in result would be wrong and the value in x would be incremented twice.

Compilation errors

Consider we are compiling this structure:

```
// Header
struct Tooltip
#if TOOLTIP_VERSION > 4
  char buffer[128];
#else
  char buffer[64];
#endif
};
```

And in C++

```
Tooltip tooltip;
memset(&tooltip, 0, sizeof(tooltip));
```

If we fail to define `TOOLTIP_VERSION` to the same value in the implementation as in the caller, then this code may stomp all over memory because it thinks the struct is 128 bytes in one place and 64 bytes in another.

Namespace issues

Macros aren't namespaced and in some cases this leads to problems where a macro definition collides with a well qualified symbol. For example code that `#include <windows.h>` gets a `#define TRUE 1`. But that excludes any other code that expects to compile on Windows from ever using `TRUE` as a const no matter how well they qualify it. Consequently code has to do workarounds such as `#undef` macros to make code work or using another value.

```
#ifdef TRUE
#define TMP_TRUE TRUE
#undef TRUE
#endif
bool value = myapp::TRUE;
#ifdef TMP_TRUE
#define TRUE TMP_TRUE
#undef TMP_TRUE
#endif
```

Ugh. But more likely we'll rename myapp::TRUE to something like myapp::MYAPP_TRUE to avoid the conflict. It's still an ugly workaround for a problem caused by inconsiderate use of macros.

Commonly used words like TRUE, FALSE, ERROR, OK, SUCCESS, FAIL are more or less unusable thanks to macros.

How Rust helps

Rust provides developers with consts, inline attributes, and platform / architecture attributes for the purpose of conditional compilation.

Rust offers macros but they consist of a set of matching rules than must generate syntactically Rust. Macro expansion is performed by the compiler so it is capable of generating errors on the macro if the macro is in error.

Type Mismatching

Consider two methods. Both are called evaluate() and they are overloaded. The main() method calls evaluate("Hello world"). What version is called in the compiled code?

```
#include <iostream>
#include <string>

using namespace std;

void evaluate(bool value) {
    cout << "Evaluating a bool " << value << endl;
}

void evaluate(const std::string &value) {
    cout << "Evaluating a string " << value << endl;
}

int main() {
    evaluate("Hello world");
    return 0;
}
```

It may surprise you to know that the bool version is called and the compiler doesn't even complain about it either:

```
Evaluating a bool 1
```

This is an example of bad type inference. A string literal (a char) *should be turned into a std::string (a C++ string has a constructor that takes char*) but the compiler chose to treat it as a bool instead.

On other occasions the compiler might spot ambiguity and complain but the blurred lines between types in C++ combined with overloading lead to errors: Here is another example where the compiler is a little more useful by generating an error, but in doing so it demonstrates the limits of overloading

```
bool evaluate(bool value);
bool evaluate(double value);
```

These overloaded methods should be distinct but they're not distinct enough as far as the compiler is concerned.

In summary, blurred and confusing rules about types in C++ can cause unexpected errors that can propagate to runtime.

How Rust helps

In Rust the functions cannot be overloaded in this manner.

Rust is also more strict about type coercion - if you have a bool you cannot pass it to a function that takes an integer.

Nor can you pass an integer of one size to a function taking an integer of another size.

```
fn print_i32(value: i32) {
   println!("Your value is {}", value);
}
let value = 20i16; // 16-bit int
print_i32(value);
```

This will yield an error:

```
error[E0308]: mismatched types
  |
7 | print_i32(value);
  |           ^^^^^ expected i32, found i16
```

You must use an explicit numeric cast to turn the value into the type the function expects:

```
print_i32(value as i32);
```

Explicit / Implicit Class Constructors

It's not just overloading that can be a mess. C++ has a bunch of rules about implicit / explicit type conversion for single argument constructors.

For example:

```
class MagicNumber {
public:
    MagicNumber(int value) {}
};

void magic(const MagicNumber &m) {
  //...
}

int main() {
    //...
    magic(2016);
    return 0;
}
```

The function `magic()` takes a `const MagicNumber &` yet we called it with `2016` and it still compiled. How did it do that? Well our `MagicNumber` class has a constructor that takes an `int` so the compiler implicitly called that constructor and used the `MagicNumber` it yielded.

If we didn't want the implicit conversion (e.g. maybe it's horribly expensive to do this without knowing), then we'd have to tack an `explicit` keyword to the constructor to negate the behaviour.

```
explicit MagicNumber(int value) {}
```

It demonstrates an instance where the default behavior is probably wrong. The default *should* be `explicit` and if programmers want implicit they should be required to say it.

C++11 adds to the confusion by allowing classes to declare deleted constructors which are anti-constructors that generate an error instead of code if they match. For example, perhaps we only want implicit `int` constructors to match but we want to stop somebody passing in a `double`. In that case we can make a constructor for `double` and then delete it.

```
class MagicNumber {
public:
    MagicNumber(int value) {}
    MagicNumber(double value) = delete;
};

void magic(const MagicNumber &m) {
  //...
}

//...
magic(2016);   // OK
magic(2016.0); // error: use of deleted function
'MagicNumber::MagicNumber(double)'
```

How Rust helps

Rust does not have constructors and so there is no implicit conversion during construction. And since there is no implicit conversion there is no reason to have C++11 style function delete operators either.

You must write explicit write "constructor" functions and call them explicitly. If you want to overload the function you can use `Into<>` patterns to achieve it.

For example we might write our `MagicNumber` constructor like this:

```
struct MagicNumber { /* ... */ }

impl MagicNumber {
  fn new<T>(value: T) -> MagicNumber where T: Into<MagicNumber>
{
    value.into()
  }
}
```

We have said here that the `new()` function takes as its argument anything that type `T` which implements the trait `Into<MagicNumber>`.

So we could implement it for `i32`:

```
impl Into<MagicNumber> for i32 {
   fn into(self) {
     MagicNumber { /* ... */ }
   }
}
```

Now our client code can just call `new` and providing it provides a type which implements that trait our constructor will work:

```
   let magic = MagicNumber::new(2016);
   // But this won't work because f64 doesn't implement the
trait
   let magic = MagicNumber::new(2016.0);
```

Poor Lifetime Enforcement

A function like is completely legal and dangerous:

```
std::string &getValue() {
  std::string value("Hello world");
  return value;
}
```

This function returns a reference to a temporary variable. Whoever calls it will get a reference to garbage on the stack. Even if it appears to work (e.g. if we called the reference immediately) it is only through luck that it does.

Our compiler will probably issue a warning for this trivial example but it won't stop us from compiling it.

How Rust helps

Rust tracks the lifetime of all objects and knows when their lifetime begins and ends. It tracks references to the object, knows when it is being borrowed (being passed to a function / scope).

It generate a compiler error if it detects any violations of its lifetime / borrowing rules. So the above code would fail to compile.

Memory Allocation

Allocated memory is memory that is requested from a portion of memory called a heap, used for some purpose and returned to the free space when it is no longer required.

In C memory is allocated and freed through a relatively simple API:

- `malloc` and `calloc` allocate memory and `free` destroys it.

However C++ also needs allocates that call the appropriate constructors and destructors so in addition to C's memory allocation functions, there are keywords for allocation / free.

- `new` / `delete` for C++ class instances
- `new[]` and `delete[]` for arrays of classes
- The above but through scoped / shared pointer classes that take ownership of the pointer and free it when appropriate.

If we fail to free / delete memory that we've allocated, the program will leak memory. If we free / delete memory we've already deallocated, the program may crash. If we free a C++ class with a C `free()` the program may leak memory because any member variables will not be destroyed properly. If we fail to call the correct constructor and destructor pair the program may leak / crash.

A cottage industry of tools has sprung up just to try and debug issues with memory leaks, crashes and so forth. Tools like Valgrind etc. specialise in trying to figure out who allocated something without freeing it.

For example, what's wrong with this?

```
std::string *strings = new std::string[100];
//...
delete strings;
```

Oops we allocated an array of strings with `new[]` but called `delete` instead of `delete[]`. So instead of deleting an array of strings we called delete on the first member. 99 of those string's destructors will never be called. We should have written:

```
delete []strings;
```

But the compiler doesn't care and so we have created a potentially hard-to-find bug.

Some of the problems with memory allocation can be mitigated by wrapping pointers with scoped or shared pointer classes. But there are even problems which can prevent them from working.

It's not a good idea to allow memory allocation to cross a library boundary. So many libraries provide new / free functions through their API. Issues about balancing calls apply to them too.

How Rust helps

During normal safe programming Rust has no explicit memory allocation or deallocation. We simply declare an object and it continues to exist until its lifetime goes out of scope (i.e. nothing refers to it any more).

This is NOT garbage collection. The compiler tracks the lifetime of the object and generates code to automatically delete it at the point it is no longer used. The compiler also knows if we enclose an object's declaration inside a cell, box, rc or similar construct that the object should be allocated on the heap and otherwise it should go on the stack.

Allocation / deallocation is only available in unsafe programming. We would not only ordinarily do this except when we are interacting with an external library or function call and explicitly tag the section as unsafe.

Null Pointers

The need to test a pointer for NULL, or blindly call a pointer that might be NULL has caused so many errors that it has even been called the billion dollar mistake

TODO

Virtual Destructors

C++ allows classes to inherit from other classes.

In some cases, such as this example, this can lead to memory leaks:

```
class ABase {
public:
  ~ABase() {}
};

class A : public ABase {
  std::string *value_;
public:
  A() : value_(new std::string) {}
  ~A() { delete value_; }
};

void do_something() {
  ABase *instance = new A();
  //...
  delete instance;
}
```

So here we allocate a pointer to A, assign it to "instance" which is of type `ABase`, do something with it and finally delete it. It looks fine but we just leaked memory! When we called "delete instance" the code invoked the destructor `~ABase()` and NOT the destructor `~A()`. And `value_` was not deleted and the memory leaked. Even if we'd used a scoped pointer to wrap `value_` it would still have leaked.

The code should have said

```
class ABase {
public:
  virtual ~ABase() {}
};
```

The compiler didn't care our code was in error. It just allowed us to leak for the sake of a missing keyword.

How Rust helps

Rust also does not use inheritance so problems like ABase above cannot exist. In Rust `ABase` would be declared as a trait that A implements.

```
trait ABase {
  //...
}

struct A {
  value: String,
}

impl ABase for A {
  //...
}
```

Rust also allows our struct to implement another trait called `Drop` which is equivalent to a C++ destructor.

```
impl Drop for A {
  fn drop(&mut self) {
    println!("A has been dropped!");
  }
}
```

It allows our code to do something during destruction such as to free an open resource, log a message or whatever.

Exception Handling / Safety

There are no hard and fast rules for when a function in C++ should throw an exception and when it should return a code. So one codebase may have a tendency to throw lots of exceptions while another might throw none at all.

Aside from that, code may or may not be exception safe. That is, it may or may not free up its resources if it suffers an exception.

Articles have been written to describe the levels of guarantees that code can aim for with exception safety.

Constructors

You may also be advised to throw exceptions in constructors because there is no easy way to signal the object is an error otherwise except to set the new object into some kind of zombie / dead state via a flag that has to be tested.

```
DatabaseConn::DatabaseConn() {
  db_ = connect();
  if (db_ == NULL) {
    throw string("The database connection is null");
  }
}

// These both recover their memory
DatabaseConn db1;
DatabaseConn *db2 = new DatabaseConn();
```

But if DatabaseConn() had allocated some memory before throwing an exception, this would NOT be recovered and so ~DatabaseConn would have to clean it up.

```
DatabaseConn::DatabaseConn() {
  buffer_ = new char[100];
  // ... exception throwing code
}

DatabaseConn::~DatabaseConn() {
  if (buffer_) {
    delete[] buffer_;
  }
}
```

But if we waited until after the exception throwing to allocate memory then maybe buffer_ is not set to NULL, so we'd have to ensure we initialised it to NULL.

```
DatabaseConn::DatabaseConn() : buffer_(NULL) {
  // ... exception throwing code
  buffer_ = new char[100];
}
```

Destructors

But you will be advised NOT to throw exceptions in destructors because throwing an exception during a stack unwind from handling another exception is fatal.

```
BadNews::~BadNews() {
    if (ptr == NULL) {
      throw string("This is a bad idea");
   }
}
```

How Rust helps

The recommended way of dealing with errors is to use the `Option` and `Result` types to formally pass errors to your caller.

For irregular errors your code can choose to invoke `panic!()` which is a little like an exception in that it will cause the entire thread to unwind. If the main thread panics then the process terminates.

A `panic!()` can be caught and recovered from in some scenarios but it is the nuclear option.

Lacking exceptions might seem a bad idea but C++ demonstrates that they come with a whole raft of considerations of their own.

Templates vs Generics

What's a template?

C++ provides a way of substituting types and values into inline classes and functions called templates. Think of it as a sophisicated substitution macro - you specify a type T in the template and this can substitute for a type `int` or something else at compile time. During compilation you'll be told if there are any errors with the type you supply. This is a very powerful feature since it allows a class to be reused for many different types.

Templates are used extensively in the C++ library, Boost and in other places. Collections, strings, algorithms and various other piece of code use templates in one form or another.

However, templates only expand into code when something actually calls the inline function. Then, if the template calls other templates, the inline code is expanded again and again until there is a large body of code which can be compiled. A small error in our code can propogate into an enormous wall of noise in the middle of some expanded template.

For example a vector takes a type it holds as a template parameter. So we can create a vector of PatientRecords.

```
class PatientRecord {
  std::string name_;

  PatientRecord() {}
  PatientRecord operator= (const PatientRecord &other) { return
*this; }

public:
  PatientRecord(const std::string &name) : name_(name) {
  }
};
...
std::vector<PatientRecord> records;
```

So far so good. So let's add a record:

```
records.push_back(PatientRecord("John Doe"));
```

That works too! Now let's try to erase the record we just added:

```
records.erase(records.begin());
```

Boom!

```
c:/mingw/i686-w64-mingw32/include/c++/bits/stl_algobase.h: In
instantiation of 'static _OI std::__copy_move<true, false,
std::random_access_iterator_tag>::__copy_m(_II, _II, _OI) [with
_II = PatientRecord*; _OI = PatientRecord*]':
c:/mingw/i686-w64-
mingw32/include/c++/bits/stl_algobase.h:396:70:   required from
'_OI std::__copy_move_a(_II, _II, _OI) [with bool _IsMove =
true; _II = PatientRecord*; _OI = PatientRecord*]'
c:/mingw/i686-w64-
mingw32/include/c++/bits/stl_algobase.h:434:38:   required from
'_OI std::__copy_move_a2(_II, _II, _OI) [with bool _IsMove =
true; _II = __gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >; _OI =
__gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >]'
c:/mingw/i686-w64-
mingw32/include/c++/bits/stl_algobase.h:498:47:   required from
'_OI std::move(_II, _II, _OI) [with _II =
__gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >; _OI =
__gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >]'
c:/mingw/i686-w64-mingw32/include/c++/bits/vector.tcc:145:2:
required from 'std::vector<_Tp, _Alloc>::iterator
std::vector<_Tp, _Alloc>::_M_erase(std::vector<_Tp,
_Alloc>::iterator) [with _Tp = PatientRecord; _Alloc =
std::allocator<PatientRecord>; std::vector<_Tp,
_Alloc>::iterator = __gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >; typename std::_Vector_base<_Tp,
_Alloc>::pointer = PatientRecord*]'
c:/mingw/i686-w64-mingw32/include/c++/bits/stl_vector.h:1147:58:
required from 'std::vector<_Tp, _Alloc>::iterator
std::vector<_Tp, _Alloc>::erase(std::vector<_Tp,
_Alloc>::const_iterator) [with _Tp = PatientRecord; _Alloc =
```

```
std::allocator<PatientRecord>; std::vector<_Tp,
_Alloc>::iterator = __gnu_cxx::__normal_iterator<PatientRecord*,
std::vector<PatientRecord> >; typename std::_Vector_base<_Tp,
_Alloc>::pointer = PatientRecord*; std::vector<_Tp,
_Alloc>::const_iterator = __gnu_cxx::__normal_iterator<const
PatientRecord*, std::vector<PatientRecord> >; typename
__gnu_cxx::__alloc_traits<typename std::_Vector_base<_Tp,
_Alloc>::_Tp_alloc_type>::const_pointer = const PatientRecord*]'
..\vectest\main.cpp:22:34:   required from here
..\vectest\main.cpp:8:19: error: 'PatientRecord
PatientRecord::operator=(const PatientRecord&)' is private
     PatientRecord operator= (const PatientRecord &other) {
return *this; }
```

If you wade through that noise to the bottom we can see the erase() function wanted to call the assignment operator on PatientRecord, but couldn't because it was private.

But why did vector allow us to declare a vector with a class which didn't meet its requirements?

We were able to declare the vector, use the std::vector::push_back() function but when we called std::vector::erase() the compiler discovered some deeply nested error and threw these errors back at us.

The reason is that C++ only generates code for templates when it is called. So the declaration was not in violation, the push_back() was not in violation but the erase was.

How Rust helps

Rust has a concept similar to templates called generics. A generics is a struct or trait that takes type parameters just like a template.

However but the type can be enforced by saying the traits that it must implement. In addition any errors are meaningful.

Say we want to write a generic function that clones the input value:

```
fn clone_something<T>(value: T) -> T {
  value.clone()
}
```

We haven't even called the function yet, merely defined it. When we compile this, we'll instantly get an error in Rust.

error: no method named `clone` found for type `T` in the current scope

```
  |
4 |   value.clone();
  |         ^^^^^
  |
  = help: items from traits can only be used if the trait is
implemented and in scope; the following trait defines an item
`clone`, perhaps you need to implement it:
  = help: candidate #1: `std::clone::Clone`
```

Rust is saying we never said what T was and because some-random-type has no method called clone() we got an error. So we'll modify the function to add a trait bound to T. This binding says T must implement Clone:

```
fn clone_something<T: Clone>(value: T) -> T {
  value.clone();
}
```

Now the compiler knows T must have implement Clone it is able to resolve clone() and be happy. Next we actually call it to see what happens:

```
struct WhatHappensToMe;
let x = clone_something(10);
let y = clone_something(WhatHappensToMe{});
```

We can clone the integer 10 because integers implement the Clone trait, but our empty struct WhatHappensToMe does not implement Clone trait. So when we compile it we get an error.

```
error[E0277]: the trait bound `main::WhatHappensToMe:
std::clone::Clone` is not satisfied
  |
8 | let y = clone_something(WhatHappensToMe{});
  |         ^^^^^^^^^^^^^^^
  |
  = note: required by `main::clone_something`
```

In summary, Rust improves on templates by TODO

Compiling generic functions / structs even when they are unused and offer meaningful errors immediately.

Allow us to bind traits to generic types to constrain what we can pass into them.

Offer meaningful errors if we violate the requirements of the trait bounds

Multiple Inheritance

C++ allows code to inherit from multiple classes and they in turn could inherit from other classes. This gives rise to the dreaded *diamond pattern*.

e.g. D inherits from B and C but B and C both inherit from A. So does D have two instances of A or one?

This can cause compiler errors which are only partially solved by using something called "virtual inheritance" to convince the compiler to share A between B and C.

i.e if we knew B and C could potentially be multiply inherited we might declare them with a virtual keyword in their inheritance:

```
class B : public virtual A {
//...
};
class C: public virtual A {
};
class D: public B, public C {
//...
};
```

When D inherits from B and C, both share the same instance of A. But that assumes the authors of A, B and C were aware of this problem arising and coded themselves with the assumption that A could be shared.

The more usual normal solution for diamond patterns is "don't do it". i.e use composition or something to avoid the problem.

How Rust helps

Rust also does not use class inheritance so problems like diamond patterns cannot exist.

However traits in Rust can inherit from other traits, so potentially it could have diamond-like issues. But to ensure it doesn't, the base trait is implemented separately from any traits that inherit from it.

So if struct D implements traits B & C and they inherit from A, then A, B and C must have impl blocks.

```
trait A {
//...
}

trait B : A {
//...
}

trait C : A {
//...
}

struct D;

impl A for D {
//...
}

impl B for D {
//...
}

impl C for D {
//...
}
```

Linker Errors

C and C++ requires you supply a list of all the .obj files that form part of your library or executable.

If you omit a file by accident you will get undefined or missing references. Maintaining this list of files is an additional burden of development, ensuring to update your makefile or solution every time you add a file to your project.

How Rust Helps

Rust includes everything in your library / executable that is directly or indirectly referenced by mod commands, starting from your toplevel lib.rs or main.rs and working all the way down.

Providing you reference a module, it will be automatically built and linked into your binary.

If you use the `cargo` command, then the above also applies for external crates that you link with. The cargo command will also check for version conflicts between external libraries. If you find your cargo generating errors about compatibility conflicts between crates you may be able to resolve them by updating the Cargo.lock file like so:

```
cargo update
```

Debugging Rust

Rust compiles into machine code the same as C and benefits from sharing the same ABI and compiler backend formats as C/C++.

So you can debug Rust in the same way as C/C++. If you built your Rust executable in a gcc compatible binary format you can just invoke gdb on it:

```
gdb my_executable
```

Rust comes with a gdb wrapper script called `rust-gdb` that loads macros which perform syntax highlighting.

Enabling backtrace

If your code is crashing because of a panic!() you can get a backtrace on the console by setting the `RUST_BACKTRACE` environment variable.

```
# Windows
set RUST_BACKTRACE=1
# Unix/Linux
export RUST_BACKTRACE=1
```

Find out your target binary format

If you are in doubt what you are targeting, you may use `rustup` to show you.

```
c:\dev\visu>rustup show
Default host: x86_64-pc-windows-msvc

stable-x86_64-pc-windows-msvc (default)
rustc 1.13.0 (2c6933acc 2016-11-07)
```

Or perhaps:

```
[foo@localhost ~]$ rustup show
Default host: x86_64-unknown-linux-gnu

stable-x86_64-unknown-linux-gnu (default)
rustc 1.13.0 (2c6933acc 2016-11-07)
```

The information will tell you which debugger you can use to debug your code.

Microsoft Visual Studio

If you have the MSVC toolchain (32 or 64-bit) or the LLVM backend will generate a .pdb file and binaries will be compatible with the standard MSVC runtime.

To debug your code:

1. Open Visual Studio
2. Choose File | Open | Project/Solution...
3. Select the compiled executable
4. Open a source file to debug and set a breakpoint
5. Click the "Start" button

GDB

GDB can be invoked directly from the command line or through a plugin / IDE. From the command line it's a

TODO

LLDB

TODO

Memory Management

The memory model of Rust is quite close to C++. Structures that you declare in Rust reside on the stack or they reside in the heap.

Stack

The stack is a memory reserved by the operating system for each thread in your program. Stack is reserved for local variables based upon their predetermined size by moving a stack pointer register forward by that amount. When the local variables go out of scope, the stack pointer reduces by the same amount.

```
// Stack allocated
double pi = 3.141592735;
{
  // Stack pointer moves as values goes in and out of scope
  int values[20] = { 0, 1, 2, 3, ... , 19, 20 };
}
```

In C-style languages it is normal for the stack in each thread to be a single contiguous slab of memory that represents the "worst case" scenario for your program i.e. you will never need any more stack than the thread allocated at start. If you do exceed the stack, then you cause a stack overflow.

Some languages support the concept of split or segmented stack. In this case, the stack is a series of "stacklets" joined together by a linked list. When the stack is insufficient for the next call, it allocates another stacklet.

The gcc can support a segmented stack, but it greatly complicates stack unwinding when an exception is thrown and also when calls are made across linker boundaries, e.g. between a segmented-stack aware process and a non segmented stack dynamic library.

Stack Overflows

The main worry from using the stack is the possibility of a stack overflow, i.e the stack pointer moves out of the memory reserved for the stack and starts trampling on other memory.

This can occur in two common ways in isolation or combination:

- Deeply nested function calls, e.g. a recursive function that traverses a binary tree, or a recursive function that never stops
- Exhausting stack by using excessive and/or large local variables in functions, e.g. lots of 64KB byte arrays.

C++

Some C++ compilers won't catch an overflow at all. They have no guard page and thus allow the stack pointer to just grow whereever memory takes it until the program is destabilized and crashes.

The gcc compiler has support segmented stacks but as described earlier not without issue.

The MSVC compiler adds a guard page and stack pointer checks when when the stack pointer could advance more than a page in a single jump and potentially miss the guard page.

Rust

Rust used to support a segmented stack as a means of detecting memory violation but since 1.4 has replaced it with a guard page at the end of the stack space. If the guard page is touched by a memory write, it will generate a segmentation fault that halts the thread. Guard pages open up a small risk that the stack could grow well in excess of the guard and it might take some time for a write to the guard to generate a fault.

Rust aspires to support stack probe code generation on all platforms at which point it is likely to use that in addition to a guard page. A stack probe is additional generated code on functions that use more than a page of space for local variables to test if they exceed the stack.

Rust reduces the risk stack overflows in some indirect ways. It's easy in C++ through inheritance or by calling a polymorphic method inadvertently set off a recursive loop

Heap

Heap is a memory that the language runtime requests from the operating system and makes available to your code through memory allocation calls

C++

```
char * v = (char *) malloc(128);
memset(v, 0, 128);
strcpy(v, "Hello world");
//...
free(string);

double *values = new double[10];
for (int i = 0; i < 10; i++) {
  values[i] = double(i);
}
delete []values;
```

Allocation simply means a portion of the heap is marked as in-use and the code is provided with a pointer to the reserved area to do what it likes with. Free causes the portion to be returned to its free state, coalescing with any free areas that it resides next to in memory.

A heap can grow and code might create multiple heaps and might even be compelled to in order control problems such as heap fragmentation.

Rust

To allocate memory on the heap in Rust you declare data inside of a a box. For example to create a 1k block of bytes:

```
let x: Box<[u8]> = Box::new([0; 1024]);
```

Many structs in std:: and elsewhere will have a stack based portion and also use use heap internally to hold their buffers.

Heap fragmentation

Heap fragmentation happens when contiguous space in the heap is limited by the pattern of memory allocations that it already contains. When this happens a memory allocation can fail and the heap must be grown to make it succeed. In systems which do not have virtual memory / paging, memory exhaustion caused by fragmentation can cause the program or even the operating system to fail completely.

The easiest way to see fragmentation is with a simple example. We'll pretend there is no housekeeping structures, guard blocks or other things to get in the way. Imagine a 10 byte heap, where every byte is initially free.

Now allocate 5 bytes for object of type A. The heap reserves 5 bytes and marks them used.



Now allocate 1 byte for object of type B. This is also marked used.



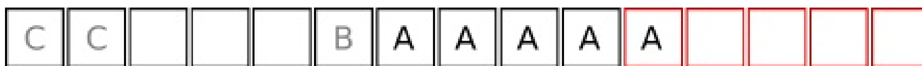
Now free object A. The the portion of heap is marked unused. Now we have a block of 5 bytes free and a block with 4 bytes free.



Now allocate 2 bytes for object of type C. Now we have a block of 3 bytes free and a block with 4 bytes free.



Now allocate 5 slots for object of type A - Oops we can't! The heap has 7 bytes free but they are not contiguous. At this point the runtime would be forced to grow the heap, i.e. ask the operating system for another chunk of memory at which point it can allocate 5 bytes for A.



The above assumes the heap is a contiguous, or that memory paging makes it seem so. On some systems, it might be that the heap is a linked list of chunks, in which case the allocated space for A would have to reside be in a single chunk, the newly allocated portion above.

This is also an exagerated example, but it demonstrates how heap can have space, but not enough to fufilly allocations without growing.

Software running in embedded devices are particularly vulnerable to fragmentation because they do not have virtual memory, have low physical memory and normally have to run for days, weeks or years at a time.

One major problem for C++ is that heap fragmentation is almost impossible to avoid. The standard template library allocates memory for virtually all string and collection work, and if a string / collection grows then it may have to reallocate more memory.

The only way to mitigate the issue is to choose the best collection, and to reserve capacity wherever possible.

```
std::vector<double> values;
values.reserve(10);
for (int i = 0; i < 10; i++) {
  values.push_back(double(i));
}
```

Rust also has this issue and strings / collections have methods to reserve capacity. But as a consequence of its design it prefers the stack over the heap. Unless you explicitly allocate memory by putting it into a Box, Cell or RefCell you do not allocate it on the heap.

RAII

RAII stands for Resource Acquisiton Is Initalization. It's a programming pattern that ties access to some resource the object's lifetime

C++ classes allow a pattern called RAII (). A class constructor acquires some resource, the destructor releases that resource. As soon as the class goes out of scope, the resource is released.

TODO C++ example

Rust is inherently RAII and enforces it through lifetimes. When an object goes out of scope, the thing it holds is released. Rust also allows the programmer to explicitly drop a struct earlier than its natural lifetime if there is a reason to.

RAII is most commonly seen for heap allocated memory but it can apply to files, system handles etc.

TODO Rust example

Rust's standard library

The core functionality in Rust is provided by a module called `std`. This is the standard runtime library.

As with its C++ namesake, everything can be referenced through a `std::` namespace prefix or via a `use std::{foo}` import.

Some of std is implicitly available by a special `std::prelude` that is automatically used (along with a reference to the std crate) without declaration. The prelude contains functionality that virtually all code is likely to use and therefore Rust spares code from having to import it:

- String and ToString trait
- Iterators traits of various kinds - Iterator, Exten, IntoIterator etc.
- Result<> and Option<> enums
- Conversion traits AsRef, AsMut, Into, From
- Vec heap allocated vector
- Other traits such as Drop, Fn, FnMut, FnOnce, Box, Clone, Copy, Send, Sized, Sync, PartialEq, PartialOrd etc.
- Macros such as println!, format!, assert! etc.

```
// You don't need these
extern crate std;
use std::prelude::*;
```

There are various sub-modules under std that concern themselves with aspects of development. Here are just some of them:

1. clone – the Clone trait
2. cmp – Eq, Ord, PartialEq, PartialOrd traits. These traits are used for equality and ordering functionality.
3. collections - contains the standard collection types for sequences, maps, sets, and miscellaneous. e.g. Vec and HashMap are members of this module.
4. env – environmental helpers - command line arguments, status codes, environment variables, temporary folder
5. fmt – utilities for formatting and printing strings
6. fs - filesystem manipulation
7. io – Read and Write traits that are implemented by streams / buffers in file system and networking, stdio functionality

8. mem – memory primitives
9. net – networking
10. path – path manipulation
11. process – spawn, fork, exec etc.

C / C++ lib to Rust lib cross reference

TODO

Note that Rust's std namespace contains a lot of stuff not in the standard C or C++ libraries and a lot of things are not directly analogous. For example the standard C / C++ library have very little to say about sockets, or path manipulation, or atomically incrementing numbers, or creating threads.

C	C++	Rust
T [S], e.g. char foo[20]	std::array (C++11)	[T; S], e.g. let foo: [u8; 20] = [0; 20]
char * or char[] with functions such as strcmp, strcpy, strstr, strdup etc. Plus wide equivalents to these.	std::string, std::wstring, std::u16string (C++11), std::u32string (C++11)	&str or String as appropriate
-	std::vector	std::vec::Vec or std::collections::VecDeque
-	std::list	std::collections::LinkedList
-	std::set	std::collections::HashSet, std::collections::BTreeSet
-	std::map	std::collections::HashMap, std::collections::BTreeMap
fopen, fclose, fread / fwrite, fseek etc.	std::ofstream, std::ifstream, std::fstream	TODO
Math functions such as cos, sin, tan, acos, asin, atan, pow, abs, log, log10, floor, ceil are defined in	-	Math functions are direction accessible from f64. f32 types., e.g. 1.0f64.cos().

Note that because due to the decimal point being used on a float, you have to prefix f32 or f64 to literals when you call them so the compiler can figure out what you're doing.

Standard Traits

Some traits are system defined and in some cases can be derived automatically.

In others they cause the compiler to generate additional code for you such as the Drop trait (described in class destructor section)

Drop

The Drop trait allows you do something when an object is dropped, such as add additional logging or whatever.

Copy

A struct implementing a Copy trait can be copied through assignment, i.e. if you assign a to b then a and b now how copies of the object, independent of each other. The Copy trait really only useful when you have small amounts of data that represent a type or value of some kind. TODO copy example, e.g. struct PlayingCard { suit: Suit, rank: Rank } If you find yourself with a type that is larger, or contains heap allocated memory then you should use clone.

Clone

A struct implementing the Clone trait has a .clone() method. Unlike Copy you must explicitly .clone() the instance to create another. TODO clone example

Eq, PartialEq

TODO equality

Ord, PartialOrd

TODO ordering

Rust Cookbook

Numbers

Convert a number to a string

Let's say you have an integer you want to turn into a string.

In C++ you might do one of the following:

```
const int value = 17;
std::string value_as_string;

// Nonstandard C itoa() (also not thread safe)
value_as_string = itoa(value);

// OR _itoa()
char buffer[16];
_itoa(value, buffer, 10);

// OR
sprintf(buffer, "%d", value);

// OR
stringstream ss;
ss << value;
value_as_string = ss.str();

// OR (boost)
value_as_string = boost::lexical_cast<std::string>(ivalue);
```

All of these have issues. Some are extensions to the standard, others may not be thread safe, some may break if `value` was changed to another type, e.g. `long long`.

Rust makes it far easier because numeric primitives implement a trait called ToString. The ToString trait has a to_string() function. So to convert the number to string is as simple as this:

```
let value = 17u32;
let value_as_string = value.to_string();
```

The same is true for a floating point number:

```
let value = 100.00345f32;
let value_as_string = value.to_string();
```

Convert a number to a string with precision / padding

In C you would add precision of padding using printf operations:

```
double value = 1234.66667;
char result[32];
sprintf(result, "%08.2d", value);
```

In C++ you could use the C way (and to be honest it's easier than what is written below), or you can set padding and precision through an ostream:

```
// TODO validate
double value = 1234.66667;
ostringstream ss;
ss << setfill('0') << setw(8) << setprecision(2) << value;
```

In Rust you can use format!() [https://doc.rust-lang.org/std/fmt/] for this purpose and it is similar to printf / sprintf:

```
let value = 1234.66667;
let value_as_string = format!("{:08.2}", value);
println!("value = {}", value_as_string);
```

Output

```
value = 01234.67
```

Convert a number to a localized string

Some locales will use dots or commas for separators. Some languages will use dots or commas for the decimal place. In order to format these strings we need to make use of the locale.

TODO

Convert a string to a number

In C / C++ a number might be converted from a string to a number in a number of ways

```
int value = atoi(value_as_str);
```

TODO

In Rust we have a &str containing a number:

```
let value_as_str = "12345";
```

Any type that implements a trait called FromStr can take its type from a string. All the standard primitive types implement FromStr so we can simply say this:

```
let value_as_str = "12345";
let value = i32::from_str(value_as_str).unwrap();
```

Note the unwrap() at the end - the FromStr::from_str() returns the value inside a Result, to allow for the possibility that the string cannot be parsed. Production code should test for errors before calling unwrap() or it will panic.

Another way to get the string is to call parse() on the &str or String itself. In this case, you use a slightly odd looking syntax nicknamed 'turbofish' which looks like this:

```
use std::str::FromStr;
let value_as_str = "12345";
let value = value_as_str.parse::<i32>().unwrap();
```

The string's implementation of parse() is a generic that works with any type implementing `FromStr`. So calling `parse::<i32>` is equivalent to calling `i32::from_str()`.

Note one immediate advantage of Rust is it uses string slices. That means you could have a long string with many numbers separated by delimiters and parse numbers straight out of the middle of it without constructing intermediate copies.

Converting between numeric types

Converting between numeric types is as easy as using the "as" keyword.

```
let f = 1234.42f32;
let i = f as i32;
println!("Value = {}", i);
```

The result in i is the integer part of f.

```
Value = 1234
```

Strings

Rust comes with some very powerful functions that are attached to every &str and String type. These mostly correspond to what you may be used to on the std::string class and in boost string algorithms.

Most find / match / trim / split string operations in Rust are efficient because they neither modify the existing string, nor return a duplicate to you. Instead they return slices, i.e. a pointer and a length into your existing string to denote the range that is the result.

It is only operations that modify the string contents themselves such as creating upper or lowercase versions that will return a new copy of a string.

Trimming a string

Spaces, tabs and other Unicode characters defined as whitespace can be trimmed from a string.

All strings have access to the following functions

```
fn trim(&self) -> &str
fn trim_left(&self) -> &str
fn trim_right(&self) -> &str
```

Note the signatures of these functions - they are not mutable. The functions return a slice of the string that excludes the leading and / or trailing whitespace removed. In other words it is not duplicating the string, nor is it modifying the existing string. Instead it is just telling you what the trimmed range is within the &str you're already looking at.

So

```
let untrimmed_str = " this is test with whitespace    \t";
let trimmed_str = untrimmed_str.trim();
println!("Trimmed str = \"{}\"", trimmed_str);
```

Yields:

```
Trimmed str = "this is test with whitespace"
```

Also be aware that trim_left() and and trim_right() above are affected by the directionality of the string.

Most strings read from left-to-right, but strings in Arabic or Hebrew are read right-to-left and will start with a control character that sets their base direction right-to-left. If that character is present, trim_left() actually trims from the right and trim_right() trims from the left.

Get the length of a string

Every &str and String has a len() function.

```
let message = "All good things come to those who wait";
println!("Length = {}", message.len());
```

Note that len() is the length in bytes. If you want the number of characters you need to call message.chars().count(), e.g.

```
let message = "文字列の長さ";
assert_eq!(message.chars().count(), 6);
```

Splitting a string

String slices and String have a variety of `split` methods that return an iterable collection of slices on a string:

```
let input = "20,30,400,100,21,-1";
let values : Vec<&str> = input.split(",").collect();
for (i, s) in values.iter().enumerate() {
    println!("Value {} = {}", i, s);
}
```

The standard `split()` takes a string pattern for the delimiter and returns a `std::str::Split` struct that is an double-ended iterator representation of the matching result. We could call the iterator directly if we so wished but the `collect()` method above puts the values of the iterator into a `Vec<&str>`.

```
Value 0 = 20
Value 1 = 30
Value 2 = 400
Value 3 = 100
Value 4 = 21
Value 5 = -1
```

A string can also be split on an index, e.g.

```
let (left, right) = "No Mister Bond I expect you to
die".split_at(14);
println!("Left = {}", left);
println!("Right = {}", right);
```

Note that index is the *byte index*! The function will panic if the index is in the centre of a UTF-8 codepoint.

Another useful function is `split_whitespace` that splits on tabs, spaces, newlines and other Unicode whitespace. Any amount of whitespace is treated as a single delimiter.

```
// Split whitespace
for s in " All good   \n\n\tthings  to those who
wait".split_whitespace() {
    println!("Part - {}", s);
}
```

Yields the output.

```
Part - All
Part - good
Part - things
Part - to
Part - those
Part - who
Part - wait
```

Tokenizing a string

TODO

Joining strings together

TODO

Getting a substring

TODO

Converting a string between upper and lower case

Strings have these functions for converting between upper and lower case:

```
fn to_lowercase(&self) -> String
fn to_uppercase(&self) -> String
```

These functions will return a new String that contains the upper or lower case version of the input. Upper and lower case are defined by Unicode rules. Languages that have no upper or lowercase strings may return the same characters.

Doing a case insensitive compare

TODO

Using regular expression matches

TODO

Date and Time

Get the current date and time

TODO time_rs

UTC

TODO explain what UTC is and why maintaining time in UTC is vital Epochs etc. TODO preamble about what an epoch is, the Unix epoch and other epochs

Setting a timer

TODO setting a timer

System time vs UTC

TODO the reason timers might be set in system uptime vs timers being set in UTC. Answer because users and NTP can change the UTC time wherease system time is relative to bootup. So setting a timer to run 10s from now will always work against system time where setting a timer to run 10s from now in UTC could fail if the OS sets time back by an hour.

Formatting a date as a string

TODO standard date formatting UTC TODO example

Parsing a date from a string

TODO parsing a date from a string's TODO example

Performing date / time arithmetic

Collections

Creating a static array

An array primitive consists of a type and a length. e.g. a 16 kilobyte array of bytes can be created and zeroed like this:

```
let values: [u8; 16384] = [0; 16384];
```

The variable specifies the type and length and the assignment operator assigns 0 to every element.

The type, length and values can be initialized implicitly in-place like this:

```
let my_array = [ "Cat", "Dog", "Fish", "Donkey", "Albatross" ];
println!("{:?}", my_array);
```

This is an array of 5 &str values. The compiler will complain if we try to mix types in the array. We could also declare the array and manipulate it:

```
let mut my_array: [&'static str; 5] = [""; 5];
// Set some values
my_array[0] = "Cat";
my_array[1] = "Dog";
my_array[2] = "Fish";
my_array[3] = "Donkey";
my_array[4] = "Albatross";
println!("{:?}", my_array);
```

Note in this case we declared the array, each element received an empty value. Then our code programmatically set the new element value. The latter form would obviously be useful for arrays that change. The latter would be useful for arrays which do not.

Creating a dynamic vector

A vector is a linear array of values. Unlike an array which has a fixed length, a vector can grow or shrink over time.

A vector can be created using the vec! macro like this:

```
let mut my_vector = vec![1984, 1985, 1988, 1995, 2001];
```

This creates a mutable Vec and prepopulates it with 5 values. Note how the vec! macro can use square brackets for its arguments. We could have used round brackets and it would have meant the same.

A new Vec can also be made using Vec::new() or Vec::with_capacity(size)

```rust
let mut my_array = Vec::new();
my_array.push("Hello");
let my_presized_array = Vec::with_capacity(100);
```

It is strongly recommended you use Vec::with_capacity() to create a vector with enough capacity for maximum number of elements you expect the vector to contain. It prevents the runtime from having to reallocate and copy data if you keep exceeding the existing capacity. It also significantly reduces heap fragmentation.

Removing values from a vector

Sometimes you want to strip out values from a list which match some predicate. In which case there is a handy function for that purpose. TODO `.retain`

Sorting a vector

A vector can be sorted by the natural sort order of the elements it contains:

```rust
let mut values = vec![ 99, -1, 3, 555, 76];
values.sort();
println!("Values = {:?}", values);
```

Sorting is done using the Ord trait and calling Ord::cmp() on the elements to compare them to each other.

Comparison can also be done through a closure and Vec::sort_by()

TODO `.sort_by` TODO `.sort_by_key`

Stripping out duplicates from a vector

Assuming your vec is sorted, you can strip out consecutive duplicate entries using dedup(). This function won't work and the result will be undefined if your vector is not sorted. TODO .dedup

Creating a linked list

A linked list is more suitable than a vector when items are likely to be inserted or removed from either end or from points within the list.

`std::collections::LinkedList`

Creating a hash set

A hash set is a unique collection of objects. It is particularly useful for removing duplicates that might occur in the input. `std::collections::HashSet`

Creating a hash map

A hash map consists of a key and a value. It is used for look up operations `std::collections::HashMap`

Iterating collections

TODO

Iterator adaptors

TODO

An adaptor turns the iterator into a new value

`.enum` `.map(X)` `.take(N)` `.filter(X)`

Consuming iterators

A consumer is a convenience way of iterating a collection and producing a value or a set of values from the result.

`.collect()`

`.find()` will return the first matching element that matches the closure predicate. TODO

`.fold()` is a way of doing calculations on the collection. It takes a base value, and then calls a closure to accumulate the value upon the result of the last value. TODO Processing collections

Localization

Unicode considerations

TODO

Externalizing strings

TODO

Building strings from parameters

TODO

Creating a localization file

TODO

Logging

Files and streams

Rust comes with two standard modules:

- std::io contains various stream related traits and other functionality.
- std::fs contains filesystem related functionality including the implementation of IO traits to work with files.

Creating a directory

A directory can be created with `std::fs::DirBuilder`, e.g.

```
let result =
DirBuilder::new().recursive(true).create("/tmp/work_dir");
```

File paths

Windows and Unix systems have different notation for path separators and a number of other differences. e.g. Windows has drive letters, long paths, and network paths called UNCs.

Rust provides a PathBuf struct for manipulating paths and a Path which acts like a slice and can be the full path or just a portion of one.

TODO simple example of a path being created

TODO simple example of a Path slice in actively

TODO simple example of relative path made absolute

Windows has a bunch of path prefixes so std::path::Prefix provides a way to accessing those.

TODO example of a path being made from a drive letter and path

Opening a file

A `File` is a reference to an open file on the filesystem. When the struct goes out of scope the file is closed. There are static functions for creating or opening a file:

```
use std::io::prelude::*;
use std::fs::File;

let mut f = try!(File::open("myfile.txt"));
TODO
```

Note that File::open() opens a file read-only by default. To open a file read-write, there is an OpenOptions struct that has methods to set the behaviour of the open file - read, write, create, append and truncate.

e.g. to open a file with read/write access, creating it if it does not already exist.

```
use std::fs::OpenOptions;

let file = OpenOptions::new()
            .read(true)
            .write(true)
            .create(true)
            .open("myfile.txt");
```

Writing to a file

TODO simple example of opening file to write

Reading lines from a file

TODO simple example of opening file text mode, printing contents

Threading

Rust actively enforces thread safety in your code. If you attempt to pass around data which is not marked thread safe (i.e. implements the Sync trait), you will get a compile error. If you use code which is implicitly not thread safe such as Rc<> you will get a compile error.

This enforcement means that Rust protects against data race conditions, however be aware it cannot protect against other forms of race conditions or deadlocks, e.g. thread 1 waits for resource B (held by thread 2) while thread 2 waits for resource A (held by thread 1).

Creating a thread

Creating a thread is simple with a closure.

TODO

Waiting for a thread to complete

TODO

Using atomic reference counting

Rust provides two reference counting types. Type Rc<> is for code residing on the same thread and so the reference counting is not atomic. Type Arc<> is for code that runs on different threads and the reference counting is atomic.

An Arc<> can only hold a Sync derived object. Whenever you clone an Arc<> or its lifetime ends, the counter is atomically incremented or decremented. The last decrement to zero causes the object to be deleted.

TODO example

Locking a shared resource

Message passing is a preferable way to prevent threads from sharing state but its not always possible.

Therefore Rust allows you to create a mutex and lock access to shared data. The guard that locks / unlocks the mutex protects the data and when the guard goes out of scope, the data is returned.

This style of guard is called TODO

Data race protection

Rust can guarantee that protection from data races, i.e. more than one thread accessing / writing to the same data at the same time.

However even Rust cannot protect against the more general problem of race conditions. e.g. if two threads lock each other's data, then the code will deadlock. This is a problem that no language can solve.

Waiting for multiple threads to finish

TODO

Sending data to a thread

Any struct that implements the Send trait is treated safe to send to another thread. Of course that applies to

Receiving data from a thread

A thread can receive messages and block until it receives one. Thus it is easy to create a worker thread of some kind.

TODO

Networking

Connecting to a server

TODO

Listening to a socket

TODO

Interacting with C

Using libc functions and types

Calling a C library

Generating a dynamic library

Calling Win32 functions

Common design patterns

Singleton

A singleton has one instance ever in your application. TODO

Factory

TODO

Observer

TODO

Facade

TODO

Flyweight

TODO

Adapter

An adapter is where we present a different interface to a client calling the adapter than the interface the code is implemented in. This might be done to make some legacy code conform to a new interface, or to manage / hide complexity which might leak out into the client.

As Rust is a relatively new language you are most likely to use an adapter pattern to wrap some existing code in C. A common use for the adapter in C++ is to wrap up a C library in RAII classes or similar.

TODO